

Mike Bach

Mobile Anwendungen mit Android

Entwicklung und praktischer Einsatz

ADDISON-WESLEY

CD

ALWAYS LEARNING

PEARSON

Mobile Anwendungen mit Android

Mike Bach

Mobile Anwendungen mit Android

Entwicklung und praktischer Einsatz



ADDISON-WESLEY

An imprint of Pearson

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

10 9 8 7 6 5 4 3 2 1

13 12

ISBN 978-3-8273-3047-5

ein Imprint der Pearson Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Einbandgestaltung: Marco Lindenbeck, webwo GmbH (mlindenbeck@webwo.de)
Lektorat: Brigitte Bauer-Schiewek, bbauer@pearson.de
Fachlektorat: Frank Biet
Korrektorat: Sandra Gottmann
Herstellung: Monika Weiher, mweiher@pearson.de
Satz: Reemers Publishing Services GmbH, Krefeld (www.reemers.de)
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland

Inhaltsübersicht

Inhaltsverzeichnis

Vorwort



Ich möchte mich gar nicht mit langen Vorreden aufhalten. Das vorliegende Buch ist entstanden, weil ich Lust drauf hatte. Lust auf dieses spannende Thema Android, Lust auf das Thema Smartphones und Tablets. Ich bin mit Freude bei der Sache und wünsche meinen Lesern dasselbe. Vielleicht kann ich mit diesem Buch dazu beitragen, dass sich Nebel lichtet, der Funke des Verstehens überspringt und die Anwendungsentwicklung für Android-Systeme einen festen Platz in Hobby und Beruf bekommt und einfach Spaß macht.

Natürlich habe ich mich total verschätzt. Ich habe den Aufwand unterschätzt, ein grottenschlechtes Zeitmanagement gehabt, und zwischendurch bin ich auch bisweilen an der Fülle des Stoffs verzweifelt. Das Gefühl kennen sicherlich einige, dass man sich zwischendrin immer wieder fragt: Wohin soll das noch führen? Hier möchte ich ausdrücklich meinen Lektorinnen Brigitte Bauer-Schiewek und Anne Herklotz ein herzliches »Danke« zurufen, weil diese beiden wirklich starke Nerven beweisen mussten, bis ich endlich auch nur in die Nähe der Fertigstellung gelangt bin.

Dann mussten einige liebe Menschen noch mehr als sonst auf mich verzichten, und ich auf sie, diejenigen wissen, wer gemeint ist, zuvorderst Sophie und Ruth und dann auch noch alle anderen, die mir wichtig sind. Ich freue mich auf die nächsten Treffen, die nächsten Feiern, die nächsten Küchenschlachten, Radfahren, Skifahren und mehr.

Meine Tochter Sophie hat mich immer sehr ermuntert, da ihr die Beispiele, die ich in diesem Buch erarbeitet habe, ziemlich gut gefallen haben. Frank Biet hat mich mit seinem fachlichen Rat konstruktiv unterstützt, vielen Dank dafür.

Ich widme dieses Buch allen Suchenden und allen Kreativen, allen Tüftlern, Künstlern und Erfindern.

Ich hoffe, dass es gut geworden ist. Manchmal denke ich, ich hätte noch mehr, noch genauer, noch tiefer schreiben müssen, meist bin ich jedoch davon überzeugt, dass es gelungen ist. Über Rückmeldungen freue ich mich, konstruktive Kritik ist mir stets willkommen. Schreibt mir an mike.bach@prisma-net.de.

Wir stehen am Anfang eines neuen, spannenden Entwicklungszyklus, und ich bin gespannt, wohin die Reise geht. Daher sollten wir auch nicht lange zögern und mit unserem Teil der Reise loslegen.

Ich wünsche allen eine gute Zeit mit diesem Buch und darüber hinaus.

Mike Bach

Zum Inhalt und zur Benutzung



Das Buch gliedert sich in vier Teile:

1. die Geschichte von und ein Überblick über Android
2. die Einrichtung der Entwicklungsumgebung
3. die Grundlagen der Anwendungsentwicklung
4. weitere, tiefer gehende Aspekte der Anwendungsentwicklung

Auf der beiliegenden CD sind die vollständigen aktuellen Downloads sowie ein Eclipse-Workspace mit den Beispielen und benutzten Bibliotheken zu finden. Ich benutze hier drei Beispiele, die jeweils zum Einsatz kommen.

Die Spielwiese ist eine Sammlung von Beispielen, mit denen verschiedene Aspekte kurz beleuchtet werden. Hier lohnt sich ein Stöbern, um verschiedene Dinge, auch für verschiedene Android-Versionen, kennenzulernen.

Das MarbleGame ist der Ansatz eines Spiels, das zum einen die Nutzung von Sensoren und zum anderen einige grafische Konzepte demonstriert. Eine zwinkernde Murmel hüpft in einer zweidimensionalen Umgebung, die aber aus drei Ebenen aufgebaut ist, durch die Gegend, die Richtung können wir durch Neigen des Geräts bestimmen. Entstanden ist es auch aus dem Wunsch, beschleunigte Bewegung physikalisch einigermaßen realitätsnah zu simulieren. Ich denke, es ist eine Inspiration für ähnlich gelagerte Spielkonzepte.

Das ScrapBook ist schließlich eine Anwendung, die zum einen die Idee des Kritzelns auf einzelnen Seiten, zum anderen das Scrapbooking, also das Sammeln und Verzieren von Fotos oder anderen Materialien zu Ereignissen an unterschiedlichen Orten, aufgreift. Im vorliegenden Stadium zeigt es die Auswertung von Gesten und Multitouch-Ereignissen, die Erstellung eigener Widgets, unterschiedliche grafische Konzepte und den Zugriff auf Kamera und das Mitschneiden von Audioaufnahmen, Geo-Tagging und noch einige Dinge mehr.

Am besten schaut man sich während des Durcharbeitens die Beispiele »live« in der Entwicklungsumgebung an und versucht, die Erklärungen anhand des Codes nachzuvollziehen. Ich denke, der Sourcecode und die schriftlichen Ausführungen machen das Buch einfach gemeinsam komplett und sollten nicht getrennt voneinander benutzt werden.

Ich persönlich gehe gern so vor, dass ich querlese und versuche, die Beispiele bzw. die Sourcecodes zu verstehen. Die Eclipse-Entwicklungsumgebung macht uns das auch leicht,

da wir mit gehaltener Strg-Taste einfach auf einen Methodennamen klicken können, um zum entsprechenden Codeabschnitt zu gelangen. So kann man sich sehr schön durch den Programmablauf und die dahinter liegenden Klassen wühlen.

Dieses Buch kann aber nur bestimmte Teile der Anwendungsentwicklung abdecken, das gesamte Android-System ist so umfangreich, dass eine umfassende Beschreibung sehr, wirklich sehr umfangreich werden würde. Daher empfehle ich, neben dem Buch und dem Sourcecode auch immer die Referenz unter `developer.android.de` im Browser offen zu halten, um die verfügbaren Klassen und Methoden in ihrer Gesamtheit weiter recherchieren zu können.

Wie bei allen Themen, die Programmierung zum Inhalt haben, macht auch hier die Übung den Meister, und der Appetit kommt bekanntlich beim Essen. Daher sollte die Eclipse mit den Beispielprojekten und eigenen Spielwiesen auch immer wieder parallel benutzt werden. Am meisten Freude bereitet es natürlich, wenn man ein echtes Android-Gerät zum Ausprobieren benutzen kann.

Ich denke, so macht das Ganze dann wirklich richtig Spaß – mir geht es zumindest so –, und jetzt geht es dann auch los.

1 Die ganze Welt in einer Hand



Einst wurde Meister Ike no Taiga von einem Schüler gefragt: »Meister, was ist das Schwierigste am Malen?«. Taiga antwortete: »Jener Bereich des Papiers, auf dem noch nichts gemalt wurde, ist das Schwierigste.«

Zen-Parabel[1]

1.1 Geschichte, Gegenwart und Zukunft

Tja, so ist das. Da sitzt man vor dem Rechner. Der Cursor blinkt. Eigentlich kann es jetzt losgehen. Da ist etwas Neues, etwas Interessantes, momentan noch eine leere Fläche, die ich mit etwas Neuem füllen will. Denn bevor ich hier überhaupt etwas schreiben kann muss ich mich mit dem Thema eingehend beschäftigen, will viel darüber lernen, eine weiße Fläche in meinem Erfahrungsschatz füllen, meinen Horizont erweitern, neue Fähigkeiten erwerben und ganz und gar in meiner Neugierde aufgehen.

Das ist die eigentliche Triebfeder für das, was ich hier tue: Ich bin unendlich neugierig, finde gerne neue Dinge heraus und will lernen wie ich diese neuen Dinge benutzen kann.

Warum also Android?

Ich bin Jahrgang 1971 und habe meine ersten Begegnungen mit Computern in den Jahren 1977 bis 1980 gehabt. Das Ganze fing an mit einem beigegrauen Telespiel mit zwei Paddel-Controllern, das man an den Fernseher anschließen konnte und auf dem wir bis zum Abwinken PONG[2] gespielt haben.

Den ersten wirklichen Computer habe ich an der FH Gießen-Friedberg gesehen, angefasst und auch damit »gespielt«. Das waren Commodore CBM 3032/8032, und auch den legendären PET 2001 konnte ich dort in Aktion erleben. Das muss so um 1979/80 gewesen sein. Mein Vater war Professor im Fachbereich Mess- und Regeltechnik, und es ist für einen Jungen in diesem Alter wirklich extrem spannend gewesen, in richtigen Laboren rumzustöbern. Und dort standen auch die altehrwürdigen, damals ziemlich modernen, CBM-Geräte.

1 Aus »Ein kleines Buch vom Zen«, 2000, arsEdition München, ISBN 3-7607-8831-9
2 Siehe auch http://de.wikipedia.org/wiki/Pong (abgerufen: 12. Januar 2011, 11:17 MEZ)

Meine ersten Schritte in BASIC bin ich dann auch auf diesen altehrwürdigen Geräten mit grünem Bildschirm gegangen. Ich liebte diese CBM-Dinger, ich finde auch heute noch, dass die Geräte ein bisschen wie Darth Vader aussahen. Vielleicht kein Zufall, dass eins der ersten Spiele, die ich auf einem CBM 8032 gesehen habe, den Ausblick aus einem X-Wing-Fighter-Cockpit bot und das Abschießen von TIE-Jägern zum Inhalt hatte.

Es ist wohl auch kein Zufall dass George Lucas durch den Film Star Wars und die Gründung der Firma *Industrial Lights and Magic* (*ILM*) der Computer- und Softwareindustrie gerade im Bereich Computergrafik, digitale Bildverarbeitung und computergestützte Bilderzeugung einige immens wichtige neue Impulse gegeben hat. Der Urahn von Photoshop z.B. wurde unter anderem von dem ehemaligen ILM-Mitarbeiter John Knoll entwickelt und für den Film *The Abbyss* eingesetzt.

Commodore hat mich dann auch nicht losgelassen, ich kann mich noch gut daran erinnern, wie wir um 1981 an einem VC 20 ein endloses DATA-Listing abgetippt haben, um dann Space Invaders in einer schier wahnsinnigen Auflösung von 176 x 184 Pixeln in ganzen zwei Farben zu spielen (wahrscheinlich waren es mehr, weil der Programmierer irgendwelche Tricks angewendet hatte).

Und dann kam er, der C64. Was gibt es dazu noch zu sagen? Von der Datasette über die erste Floppy (mit selbst gelochten Disketten – und zwar *nicht* zum Abheften), einem Elektronikkurs an der Volkshochschule, Simons-Basic mit Balken- und Tortendiagrammen, dem ersten Pascal-Compiler, der Zeitschrift »64'er«[3] und »INPUT 64«[4] auf Datenträger hat meine computertechnische Sozialisation mit diesem Brotkasten stattgefunden.

Ich habe »Space Taxi«, »Ball Blazer« und »Boulder Dash« geliebt.

Danach folgten Schlag auf Schlag der Commodore PC 10 und PC 20 (der erste mit Festplatte, ich glaube unglaubliche 10 MB), meine erste integrierte Entwicklungsumgebung war Turbo-Pascal 1.0. auf einem Commodore PC 10 mit Floppy und 512 KB Hauptspeicher. Parallel einer der ersten Amigas, es hat unglaublich viel Spaß gemacht, die Entwicklung der Hardware, der Betriebssysteme und der Entwicklungsumgebungen recht hautnah mitzuerleben.

Ich hatte Tränen in den Augen als die Firma Commodore vom Markt verschwand.

Danach kamen dank der Standardisierung und der Verbreitung von MS-DOS (wobei hier von »Dank« zu sprechen teils ironisch zu verstehen ist), später Windows, beliebig austauschbare Personal Computer, die für mich keinen Kultstatus und keine Seele mehr besaßen und nur noch Mittel zum Zweck wurden. Da kann ich die Apple-Fans schon verstehen, hat es Steve Jobs doch meiner Meinung nach gut verstanden, genau das seinen Geräten und seiner Software zu bewahren: den Kultfaktor.

3 Markt&Technik Verlag
4 Heise Verlag

Was aber bleibt von dieser Geschichte? Das Faszinierende war, dass diese Zeit eine Pionierzeit und eine tolle Zeit für Entdecker war. Die Systeme waren so offen, dass man viel selber machen konnte und selber machen musste. Ich schätze, dass es so ähnlich war, als weit vor meiner Zeit das Radio auf den Markt kam.

Parallel dazu hat sich die Telekommunikation weiterentwickelt. Ich kann mich noch dunkel an das Geräusch der Wählscheibe eines grauen Telefons erinnern. Was war das ein Fortschritt, als wir das erste Telefon mit Tasten bekamen. Ich denke, es muss dann Ende der 80er gewesen sein, dass ich das erste Mal ein Autotelefon gesehen habe, und Mitte der 90er durfte ich dann bei meinem ersten Arbeitgeber ein abgelegtes C-Netz Telefon in mein Auto einbauen. Danach ging auch das Schlag auf Schlag, und die Telefonknochen haben sich zu Geräten weiterentwickelt, bei denen man aufpassen muss dass man sie beim Telefonieren nicht einatmet und verschluckt.

Was leider ein bisschen an mir vorbeigegangen ist, ist die Verbindung beider Welten, der Computer und der Telekommunikation. Ich war eher der einsame Softwareentwickler mit einer Leidenschaft für Grafik und Datenbankanwendungen, die Mailboxszene und die Entwicklung von Datex-P, ISDN etc. habe ich nur passiv mitverfolgt. Netzwerktechnik habe ich hauptsächlich im Firmenumfeld auf Novell-Basis kennengelernt, und da war meistens innerhalb der Firmen Schluss. Das hat sich erst in der ersten Hälfte der 90er verändert, als Netze per ISDN zusammengeschlossen wurden, ein Kollege mir seine Mailbox-Tätigkeiten (teils noch über Akustikkoppler) gezeigt hat und das Internet mit CompuServe und AOL Mitte der 90er anfing massentauglich zu werden.

Ab 1998 habe ich mich dann intensiv mit dem Phänomen Internet auseinandergesetzt, meine ersten Domainanmeldungen durchgeführt, die ersten eigenen IP-Adressen beantragt und den ersten eigenen Web- und Mailserver aufgebaut. Mit dem World-Wide-Web bzw. dem Internet hat sich eine ähnliche Entwicklung vollzogen wie bei den Computern. Und auch hier hat es eine Zeit gegeben, in der viel zu entdecken war und echte Pionierarbeit geleistet werden konnte. Und langsam, mit fortschreitender Miniaturisierung der Computertechnik, der Verbreitung des Mobiltelefons und der entsprechenden Infrastruktur und mit fortschreitender Vernetzung der Welt über das Internet konnten neue weiße Flächen gefüllt und neue Ideen entwickelt werden: Geräte zu bauen, die handlich und ständig einsatzbereit sind, mit denen zu jeder Zeit kommuniziert, zu jeder Zeit Musik gehört, Fernsehen und Videos geschaut, Bücher, Zeitschriften und Zeitungen gelesen und auch noch gespielt werden kann. Geräte, die uns zumindest hinsichtlich Kommunikation und Informationsfluss die Welt in unserer Hände legen.

So verschmelzen in den heutigen Smartphones und den neuen Tablet-Computern die Entwicklungen der letzten knapp 200 Jahre (ausgehend von der Vernetzung über den Morsetelegraphen) im Bereich Telekommunikation, Netzwerktechnik und Computertechnik und eröffnen wieder neue Anwendungsmöglichkeiten und damit natürlich auch neue Märkte.

1.2 Warum also Android?

Für mich war und ist immer das Interessanteste: Wie funktioniert das? Wie kann ich es selbst machen? Was kann ich damit machen? Ich bin weniger Anwender, sondern mehr Entdecker und Macher. Mein Hauptinteresse gilt dabei der Software, die Seite der Elektronik hat mich immer nur am Rande interessiert. Was mich schon immer umgetrieben hat, ist die Faszination für Grafik (Benutzeroberflächen, Präsentation, Animation), Interaktion (Steuerung und Benutzeroberflächen) und Sammeln und Speichern von Informationen (Fotografie/Videoaufnahme, Sprache/Musik, Signale, Datenübertragung und Netzwerk, Datenbanken).

Außerdem träumte ich schon länger von einem Gerät das nicht ewig Zeit benötigt, bis es betriebsbereit ist (»Hochfahren«), das ich einfach mal auf den Küchentisch legen kann und das handlich und bequem zu transportieren ist. Ein Gerät, mit dem man schnell mal E-Mails checken kann, mit dem man eben schnell mal kommunizieren kann, eben schnell mal etwas nachschlagen kann, das in der Bedienung Spass macht und mit dem man noch andere, vielleicht auch verrückte Sachen wie spielen, navigieren, geocachen, skizzieren kann.

Und auf einmal gibt es diese Geräte, und ich muss neidlos/neidvoll anerkennen, dass es Apple war, die hier einen Riesenschritt nach vorne gemacht haben. Wobei man die Entwicklung der E-Books, die frühen Tablet-PCs und auch die Entwicklung von Subnotebooks/Netbooks nicht vergessen darf.

Aber die konsequente Ausrichtung des iPhones und des iPads auf den Lustfaktor hatte diesen Geräten einfach das Entscheidende voraus: cool, schick, anschmiegsam, neu, amazing.

Aber: Apple liefert alles aus einer Hand und behält sich konsequenterweise die Kontrolle über Hardware, Betriebssystem und Software und auch die Entwicklungsumgebung vor. Darüber hinaus, mit dem genialen i-Tunes-Store und dem App-Store-Prinzip, kontrolliert Apple auch noch, was an Inhalten auf das Gerät kommt und was nicht.

Das hat unbestreitbare Vorteile: Das System wird dadurch relativ stabil und betriebssicher, weil nicht Hinz und Kunz einfach so Inhalte und Software für das Gerät publizieren können.

Ich empfinde das aber für mich und meine Anforderungen als Nachteil. Ich brauche auf meinem Gerät (relativ) uneingeschränkten Zugriff, und die Hürde, Software für das Gerät zu entwickeln, soll (relativ) niedrig sein. Außerdem bin ich ein Verfechter davon, dass auf mein Gerät die Inhalte und Programme sollen, die ich bestimme, und deren Nutzung nicht durch den Hersteller des Geräts gefiltert oder eingeschränkt werden kann. Um hier der Diskussion vorzubeugen: Mir ist natürlich bewusst, dass auch bei Android Google einen gewissen beschränkenden oder reglementierenden Zugriff ausüben kann. Aber dennoch, und das liegt, denke ich, auch in der Historie und dem Aufbau von Android begründet, ist meine Freiheit hier ungleich größer als bei den Apple-Produkten.

Android ist deshalb für mich auch attraktiv weil das gesamte System als Open-Source- Projekt konzipiert ist. Dadurch sind die Quellen offen gelegt, und ich kann, wenn ich will, bis in den hintersten Winkel des Betriebssystems stöbern. Außerdem ist der Zugang zu Informationen ohne irgendwelche Anmeldungen von Entwicklerkonten oder Ähnlichem möglich. Eine einmalige, kostenpflichtige Anmeldung ist erst dann notwendig, wenn die eigenen Ap-

plikationen auf dem Android-Market angeboten werden sollen. Während der Entwicklung benötigt man das allerdings nicht, denn die Applikationen können in den Emulatoren und auch ohne Probleme auf einem echten Android-Gerät getestet werden.

Und, für mich einer der Hauptgründe, mich mit Android zu beschäftigen: Die Anwendungsprogrammierung wird fast ausschließlich in Java durchgeführt, und die Android-Klassenbibliothek hat einen glasklaren, gut dokumentierten und verständlichen Aufbau. Darüber hinaus liefert die Klassenbibliothek den Zugriff auf nahezu alles, was das Gerät mitbringt, und das System ist so offen, dass man sogar seine eigene Benutzeroberfläche auf einem Android-Gerät realisieren kann. Das haben z.B. HTC mit der Oberfläche Sense oder Samsung mit der Oberfläche Touchwiz ausgenutzt.

Da ich von der Pascal- und C++-Seite komme und auch schon lange Jahre Java zur Entwicklung von Webapplikationen einsetze, lag es also sehr nahe, Android genauer anzuschauen. Im Gegensatz dazu hätte die Einarbeitung in Objective-C, das auf iOS (also iPhone, iPad) benutzt wird, für mich einen zu großen zeitlichen Aufwand bedeutet (obwohl auch Objective-C und Apples Cocoa-Bibliothek extrem interessante Technologien sind).

Java als Sprache für die Anwendungsentwicklung zu nutzen, erscheint mir als wirklicher Vorteil. Als konsequent objektorientierte Sprache führt Java einige Konzepte ein, die die Entwicklung gegenüber z.B. C++ stark vereinfachen. Trotzdem ist Java auch für C++-Entwickler, aber auch für Pascal-Entwickler oder Nutzer ähnlicher Sprachen sehr schnell zu erlernen. Und alle, die sich mit den Smartphones und Tablets zum ersten Mal mit Softwareentwicklung, oder überhaupt zum ersten Mal mit Softwarenetwicklung beschäftigen, finden in Java eine tolle Lernsprache. Ich würde es sogar fast mit Pascal vergleichen wollen, das in den 80ern und 90ern als Lernsprache schlechthin galt.

Die Tools und das SDK (Software Development Kit) sind darüber hinaus kostenlos verfügbar und können mit jeder Java-Entwicklungsumgebung genutzt werden. Ich bin ein großer Fan der Entwicklungsumgebung Eclipse, und es gibt ein Android-Plug-in um die Entwicklung von Android-Programmen innerhalb von Eclipse sehr komfortabel zu gestalten.

Natürlich ist neben all diesen Gesichtspunkten auch die zukünftige Marktentwicklung ein gewisses Entscheidungskriterium. Aus dem Artikel über Android auf Wikipedia geht hervor:

INFO

»Weltweit wurden bisher acht Millionen Android-Smartphones verkauft [...]. Am 11. Dezember 2010 gab Google an, dass 300.000 Android-Mobiltelefone pro Tag ausgeliefert werden (nach 60.000 im Februar 2010, 100.000 im Mai 2010, 160.000 im Juni 2010 und 200.000 im August 2010). Durch die Entscheidung von Google, sein Betriebssystem Herstellern von Endgeräten kostenlos zur Verfügung zu stellen, wird die Verbreitung von Android weiter gesteigert. Besonders der hohe Anteil an kostenlosen Applikationen macht den Android-Market für die Verbraucher attraktiv. [...] Nach einer Studie des Marktforschungsinstituts Nielsen Anfang Oktober 2010 ist Android seit Mitte 2010 das meistverkaufte Smartphone-Betriebssystem in den USA und hat in der Zwischenzeit einen Marktanteil von 32% bei den Neuverkäufen.«[5]

5 http://de.wikipedia.org/wiki/Android_(Betriebssystem) (abgerufen 12. März 2011, 13:40 MEZ)

Legt man diese Entwicklung zugrunde und beobachtet auch ein wenig den Herstellermarkt, kann man davon ausgehen, dass Android-Geräte einen attraktiven Marktanteil haben und einen noch attraktiveren Marktanteil haben werden.

Dazu kommt noch der folgende Sachverhalt: Bis zur Version Android 2.3 »Gingerbread« (Dezember 2010) war Android auch noch ausschließlich für Smartphones gedacht. Alle Tablets, die mit Android 2.0 bis Android 2.2 bestückt sind (z.B. Samsung Galaxy Tab), haben eigene Erweiterungen der Hersteller erfahren (übrigens auch ein Vorteil des Open-Source-Projekts), um das Betriebssystem auch auf Tablets laufen zu lassen.

Das hat sich mit der Version 3.0 »Honeycomb« geändert. Diese Version ist für Tablets gedacht, die aktuelle Version 2.3.3 weiterhin für Smartphones. LG, Motorola und Samsung haben bereits Honeycomb-Tablets auf der CES (Consumer Electronic Show) in Las Vegas im Januar 2011 und auf dem MWC (Mobile World Congress) in Barcelona im Februar 2011 vorgestellt.

Damit wird Android auch auf dem Tablet-Markt, der zurzeit vom Apple iPad dominiert wird, eine ernstzunehmende Größe. Vom iPad wurden bis Februar 2011 ca. 15 Millionen Stück abgesetzt (Q4 2010: 4.190.000[6]), vom Samsung Galaxy Tab im ersten Verkaufsmonat (Oktober 2010) 600.000 Stück[7]. Der Branchenverband BITKOM prognostiziert einen Marktanteil von Tablet-PCs in 2011 von 10% in Deutschland (gegenüber Notebooks, Netbooks und stationären PCs) sowie einen Absatz in Deutschland von 1,5 Millionen in 2011 und 2,2 Millionen in 2012[8].

Ich finde es in diesem Zusammenhang noch bemerkenswert, dass Sony Ericsson das Xperia Smartphone mit einem Gamepad ausstattet und damit quasi eine Spielkonsole auf Basis des Android-Systems herausbringt.

Schauen wir uns die Randbedingungen von Android in Summe noch einmal an:

1. Open-Source Projekt (die meisten Komponenten stehen unter der Apache 2.0-Lizenz)
2. Anwendungsentwicklung in Java
3. Einfacher Zugang zu Entwickler-Tools und SDKs
4. Geringe Investitionskosten zum Aufbau der Entwicklungsumgebung (lediglich Zeit und ggf. ein Android-Gerät – und natürlich dieses Buch)
5. »Elegante« Struktur des SDK (Klassenbibliothek, Architektur)
6. Zugriff auf nahezu alle Geräte- und Betriebssystemkomponenten
7. Entwicklung mit Eclipse (ebenfalls frei verfügbar)

---

6 http://www.apple.com/de/pr/library/2010/10/18results.html (abgerufen 12. März 2011, 14:00 MEZ)
7 http://de.wikipedia.org/wiki/Tablet-Computer (abgerufen 12. März 2011, 14:00 MEZ)
8 http://www.bitkom.org/files/documents/Download_Tablet_PC_Absatz.jpg, http://www.bitkom.org/de/markt_statistik/64086_67058.aspx (abgerufen 12. März 2011, 14:07 MEZ)

8. Steigender Marktanteil von Android-Smartphones
9. Günstige Prognose für den Absatz von Tablet-Computern

Meine Schlussfolgerung: darum Android.

1.3 Es gibt für alles eine App

Die Entscheidung ist gefallen. Ich will mich mit Android beschäftigen, etwas über das System und die Programmierung lernen und selbst Anwendungen erstellen. Die sollen dann ggf. auch vermarktet und verteilt werden.

Anwendungen heißen auf Englisch *Application*, abgekürzt *App*. Eigentlich bezeichnet der Begriff *Application* Anwendungsprogramme aller Art, also auch Anwendungen, die auf normalen PCs oder Netbooks laufen, es hat sich aber eingebürgert, vorzugsweise Anwendungen für Smartphones als *App* zu bezeichnen. Spricht also jemand von einer *App*, dann meint er oder sie in der Regel eine iPhone oder Android-Applikation, respektive eine Applikation für andere Smartphone-Betriebssysteme. Ich weiß nicht genau, wann und warum sich die Abkürzung etabliert hat, aber es fällt wohl mit dem Start des iPhones und des Apple App-Stores zusammen sowie mit dem Werbeslogan »Es gibt für alles eine App«.

Betrachtet man sich die Entwicklungsgeschichte des Computers und die Veränderungen, die sich nun durch die neue Gerätegeneration und die Verschmelzung mit dem allgegenwärtigen Internet ergeben, muss man auch die Veränderungen betrachten die sich hinsichtlich der Anwendungsprogramme vollziehen. Dabei sind einerseits die Art und Weise, wie und wofür Anwendungen entstehen, und andererseits die Verteilung und Vermarktung von Anwendungen interessant. Und natürlich auch die Betrachtung, wem es was nutzt.

Ganz früher, in der Steinzeit, als es nur Großrechner gab, waren Softwarelizenzen noch unbekannt, denn in der Regel kauften die Firmen die Rechner inklusive der Anwendungen, oder die Anwendungen wurden selbst geschrieben. Bis in die 70er-Jahre hinein wurden Software und Hardware noch nicht wirklich als getrennte Einheiten angesehen, das bedeutete aber auch, dass man die Software meist nur vom Hersteller der Hardware bezog. Offiziell wurde die Einzelhaftigkeit von Software, damit auch die Behandlung als eigenständiges Wirtschaftsgut, erst in den 70er-Jahren durch eine Entscheidung der US-Regierung anerkannt, die IBM dazu zwang, Hardware und Software in Rechnungen getrennt aufzuführen. Erst danach entstanden tatsächlich unabhängige Softwarefirmen die nur Software herstellten und vermarkteten. Der unabhängige Softwarehersteller bzw. Verkäufer von (Standard-)-Software stellte damals eine echte Neuerung dar. In den USA gehörte dazu z.B. Microsoft, in Deutschland die Firma SAP.

Wenn man über den Verkauf von Software spricht, meint man in der Regel die Überlassung der Nutzungsrechte an der Software. Ein tatsächlicher vollständiger Verkauf einer Software kommt eigentlich nur dann vor, wenn Unternehmen gekauft werden oder wenn ein Unternehmen für ein anderes Unternehmen eine Software im Auftrag programmiert und der Käufer bzw. der Auftraggeber auch die Weiterverbreitungsrechte erwirbt. Wenn man

also heute sagt, ich kaufe eine Software, dann kauft man das Recht zur Nutzung der Software. Wird Software als Freeware kostenlos angeboten, so hat man das Recht, die Software zu nutzen, ohne dass man für dieses Recht zahlen muss. Man hat aber kein Recht, diese Software weiterzugeben. Oft wird Open-Source bzw. freie Software mit Freeware verwechselt, das ist aber etwas grundsätzlich anderes. Freie Software/Open-Source-Software darf von jedem beliebig genutzt, verändert und weiterverbreitet werden, wobei es bestimmte Rahmenbedingungen gibt, unter denen das geschehen muss, z.B. unter der Nennung der ursprünglichen Autoren, der Verpflichtung, die Veränderung unter derselben Lizenz zu veröffentlichen, Veränderungen in das Ursprungsprojekt zurückzuführen und anderes. Open Source heißt aber nicht automatisch, dass man kein Geld dafür verlangen dürfte. Je nach Lizenzmodell kann man durchaus für eine Distribution (fertig gepackte und ausführbare Software), Wartung und Weiterentwicklung Geld verlangen.

Vollkommen unberührt von all dem ist das Urheberrecht. Der Urheber der Software bleibt der Urheber, und dieses Recht kann weder übertragen noch veräußert noch gekauft werden.

Die Entwicklung von Software hat sich im Laufe der Zeit gewandelt. Waren es früher noch die Spezialisten, die Software für komplette Problemlösungen und Anwendungsfälle erstellt haben, gesellten sich mit zunehmender Erschwinglichkeit von Computern und dem Aufkommen der integrierten Entwicklungsumgebungen immer mehr Entwickler und Programmierer hinzu, die sich um Spezialfälle kümmerten, einfach nützliche Werkzeuge entwickelten oder ganz neue Ideen umsetzten. So entstanden auf der einen Seite Anbieter für Standardsoftware wie Microsoft für Office-Anwendungen wie Textverarbeitung und Tabellenkalkulation und andere Programme, SAP im Bereich Unternehmenssoftware oder Anbieter für CAD-Programme etc. Auf der anderen Seite entstanden Softwarehäuser, die Anwendungen im Kundenauftrag entwickeln. Und es entstand eine Bewegung, die freie Software zur freien Nutzung für alle entwickelt, wie z.B. Linux, der Apache Webserver, die SQL Datenbank Firebird und eben auch Android.

Mit der Verbreitung des PC auch im Privatbereich, der Verbreitung des Internets, der Smartphones und Tablets hat sich der Markt für Software seit Ende der 90er extrem vergrößert. Und mit der Verbreitung der Smartphones, die die Verbreitung von Computern deutlich überholen wird bzw. bereits überholt hat, ist nun jeder Smartphone-Besitzer potenzieller Nutzer mobiler Softwareanwendungen.

Das ist natürlich für Softwareentwickler ein extrem attraktiver Markt, kann man doch theoretisch eine Menge Kunden erreichen. Es stellt sich dann aber schnell die Frage, wie erreiche ich diese Kunden und was sind die Kunden bereit, für meine Anwendung zu zahlen.

Wenn Firmen spezielle Software kaufen wollen oder müssen, die für den Betrieb wichtig ist, sind diese bereit, entsprechend Geld zu bezahlen. Je spezieller, betreuungsintensiver und unternehmenskritischer die Software, umso eher wird auch Geld dafür in die Hand genommen. Wenn ich als Entwickler bzw. Softwarehaus dann auch noch etwas in petto habe, was andere nicht haben, dann kann man gutes Geld damit verdienen. Dieses Bewusstsein der Firmen und die Bereitsschaft, Geld auszugeben ist beim privaten Endkunden allerdings ge-

rade in puncto Software und kostenpflichtiger Inhalte nicht (mehr) in dem Maße vorhanden, denn im Internet gibt es irgendwo immer etwas kostenlos. Es ist auch offensichtlich, dass ein Endkunde nicht bereit ist, für ein Einkaufzettelprogramm mehrere Euro hinzulegen, er benötigt ja auch noch ein Notizprogramm, ein Zeichenprogramm, zehn bis fünfzig Spiele und vieles mehr.

Das ist eine gewaltige Herausforderung, muss doch der gemeine Softwareentwickler zum einen die Kunden erreichen und zum anderen auch von etwas leben. Da kann ein Partner wie Googles Android-Market oder Apples App-Store eine gute Lösung sein, vor allem mit der eigentlichen Kernkompetenz der Partner im Rücken. Durch die Verbreitung der Smartphones und Tablets und die Kopplung der Geräte an die App-Stores erreichen wir eine riesige Anzahl von Anwendern, ohne dass wir, in der Theorie, einen hohen Vertriebsaufwand haben. Weiterhin übernimmt der Store-Betreiber die Abrechnung und die Auslieferung. Dadurch können wir die Anwendung günstiger anbieten und auf eine hohe Verkaufszahl hoffen. Oder wir können, und das macht einerseits den Charme, aber andererseits auch ein Problem der App-Stores aus, durch Werbung mitverdienen, indem wir über unsere App Werbung transportieren, die wiederum vom Store-Partner zur Verfügung gestellt wird.

Jetzt kann man sich auf den Standpunkt stellen, dass Werbung großer Mist ist. Aber Werbung ist eine Einnahmequelle sowohl für den Betreiber der Vermarktungsplattform als auch für den Entwickler, der Anwender selbst bräuchte für die Anwendung nichts zu zahlen. Das würde die Hemmschwelle, die Anwendung zu nutzen, verringern, der Kunde ist eher bereit, vielleicht auch unsere Anwendung auszuprobieren.

Ich denke, der Mittelweg könnte eine Lösung sein. Die Werbung muss entsprechend unauffällig und seriös sein, und man sollte eine werbefreie Version anbieten, die der Anwender bei Gefallen zu einem günstigen Preis kaufen kann.

Was aber noch viel wichtiger ist: Man muss sich als Entwickler noch viel genauer die Frage stellen, was genau der Anwender eigentlich benötigt. Da es ja für alles eine App gibt, hat man nur zwei Möglichkeiten. Erstens etwas komplett Neues zu schaffen oder zweitens etwas Bestehendes besser und ggf. günstiger anzubieten. Dabei ist es hilfreich, die Charakteristiken und Möglichkeiten des Geräts und des Betriebssystems zu kennen sowie über eine gehörige Portion Fantasie und Mut zu verfügen, seine Ideen umzusetzen.

Ein Vorteil für die Entwicklung auf den Smartphones und Tablets ist, dass es auch gar nicht mehr um komplette Anwendungspakete geht. Hier haben wir die Chance, uns auf kleinere überschaubare Einheiten zu konzentrieren, mit denen der Anwender genau eine Aufgabe, diese aber schnell, mobil und bequem lösen kann.

1.4 Die Geschichte von Android

Android wird in der öffentlichen Wahrnehmung hauptsächlich mit Google in Verbindung gebracht. Richtig ist, dass Google die Firma Android Inc. im Jahr 2005 gekauft hat. Richtig ist auch dass man vorher nicht viel über Android Inc. gehört hat. Da nun Google nicht vor Kritik

gefeit ist und das Motto »Don't be evil« bereits gehörige Kratzer bekommen hat, sollte man sich doch mal genauer anschauen auf was man sich da einlässt.

Android Inc. wurde im Oktober 2003 in Palo Alto unter anderem von Andy Rubin gegründet mit dem Ziel, intelligentere mobile Geräte zu entwickeln, die mehr über die Vorlieben und den Aufenthaltsort ihres Besitzers wissen. (Im Original: *»To develop [ ] smarter mobile devices that are more aware of its owner's location and preferences.«*[9]) Alles was man von Android Inc. wusste, war, dass die Firma an einer Software für Mobiltelefone arbeitet.

Interessant ist, dass Andy Rubin seine Karriere 1989 bei Apple als Software Engineer begann und dann bei General Magic an einem Betriebssystem für mobile Endgeräte arbeitete. General Magic ist deshalb bemerkenswert, weil die Gründer früher ebenfalls bei Apple gearbeitet haben und schon sehr früh (1990) die Vision hatten, dass klassische Anwendungen und Unterhaltungselektronik zusammenwachsen müssten. Also genau das, was heute in den Smartphones passiert und in großem Umfang mit Apples iPod begonnen hat. Allerdings war dem Konzept zu dieser Zeit noch kein Erfolg beschieden da die Geräte noch nicht leistungsfähig genug waren und die Firma General Magic statt auf das Internet auf einen proprietären Netzbetreiber gesetzt hatte.

Rubin gründete 2000, nach einer weiteren Anstellung bei Artemis Research (später WebTV), das Unternehmen Danger Inc. Das bedeutendste Produkt dieser Firma war das T-Mobile Sidekick, ein Smartphone, das auch über PDA-Funktionalität verfügte. Danger Inc. wurde 2008 von Microsoft übernommen und machte 2009 von sich reden, als durch einen Fehler im Datacenter, auf dem Sidekick-Kunden ihre Adressbücher, Kalender und Fotos liegen hatten, alle Daten verloren gingen. Rubin war da allerdings schon lange bei Google. Diese Episode ist allerdings nicht unwichtig, da solche (und andere) Ereignisse bei einer Diskussion über den momentanen Hype der »Cloud« berücksichtigt werden müssen. Viele Firmen versuchen die Kunden davon zu überzeugen, sowohl Anwendungen aus der »Wolke« zu beziehen als auch Daten in der »Wolke« zu speichern. Die »Wolke« ist eigentlich nur die riesige Rechenkapazität der Datacenter, die Firmen wie Google, Amazon, Microsoft, Facebook etc. bereits für ihre Anwendungen und Dienste aufbauen mussten. Diese Kapazitäten soll der Kunde nutzen mit dem Vorteil, selbst weniger Applikations- und Speicherinfrastruktur betreuen zu müssen, aber mit dem Nachteil, dass seine Daten nicht mehr in seiner Hand liegen und er Abhängig von einem Anbieter wird.

Der Anwender muss also sensibilisiert werden, was mit seinen Daten z.B. auf seinem Smartphone oder bei seinen im Web genutzten Diensten passiert.

2005 wurde Android Inc. von Google aufgekauft. Andy Rubin und weitere wichtige Mitarbeiter verblieben bei der Firma. Das Team um Andy Rubin begann dort mit der Entwicklung einer Softwareplattform für mobile Geräte auf Basis eines Linux-Kernels.

9 http://en.wikipedia.org/wiki/Android_%28operating_system%29#Android_Inc._founded_in_2003 (abgerufen 12. März 2011, 16:32 MEZ)

Am 5. November 2007 wurde die Open Handset Alliance (OHA) von 34 Firmen unter Googles Federführung gegründet. Ziel der Allianz ist die Entwicklung von offenen Standards für mobile Geräte. Am gleichen Tag wurde Android offiziell angekündigt, eine erste Version des SDK wurde am 12. November 2007 veröffentlicht.

Als erstes Android Smartphone war das T-Mobile G1 (aka HTC Dream) ab dem 22. Oktober 2008 erhältlich

Heute hat die OHA 80 Mitglieder.

Android wurde ab dem 21. Oktober 2008 unter einer Open-Source-Lizenz freigegeben. Google hat den kompletten Source-Code, inklusive des Netzwerk- und Telefonstacks, unter eine Apache-Lizenz gestellt. Dadurch kann jeder Gerätehersteller Android kostenlos auf sein Gerät portieren, anpassen und verändern. Aber: Android selbst ist ein Warenzeichen bzw. Handelsmarke von Google. Ein Gerätehersteller darf sein Gerät erst dann als Android-Gerät bezeichnen, wenn das Gerät von Google zertifiziert wurde.

Die erste Version des SDK mit der Versionsnummer 1.0 und ohne Bezug auf eine Süßspeise wurde am 23. September 2008 veröffentlicht. Und so ging es dann weiter:

VERSION	API LEVEL	VERÖFFENTLICHUNG
1.0	1	23. September 2008
1.1	2	09. Februar 2009
1.5 Cupcake	3	30. April 2009
1.6 Donut	4	15. September 2009
2.0/2.1 Éclair	5/6 (2.0.1) /7 (2.1)	26. Oktober 2009
2.2 Froyo (Frozen Yogourth)	8	20. Mai 2010
2.3/2.3.3 Gingerbread	9/10	6. Dezember 2010
3.0 Honeycomb	11	22. Februar 2011

Tabelle 1.1: **Veröffentlichung der Android-Versionen**

Ab der Version 1.5 erhalten die Versionen noch den Namen einer Süßspeise als Codename. Mit läuft schon jedes Mal das Wasser im Mund zusammen wenn ich mich mit der Versionshistorie beschäftige.

Für die Entwicklung ist der *API-Level* interessanter als der Name und die Versionsnummer. Der *API-Level* ist in der *Framework-API* verankert und verrät somit, welche Version des Android-Frameworks auf einem Gerät aktiv ist. Wenn wir nun Applikationen entwickeln, legen wir über eine Einstellung den API-Level fest und zeigen damit an, welche Version des Frameworks wir voraussetzen. Dem API-Level kommt damit eine besondere Bedeutung zu, um durch das Gerät dem Benutzer die Anwendungen im Android-Market präsentieren zu können, die zu seinem Gerät passen, bzw. um bei der Installation direkt prüfen zu können, ob die Anwendung auch für das Gerät geeignet ist.

1.5 Die äußere Hülle

Google stellt unter `http://source.android.com/compatibility/index.html` das CDD zur Verfügung, das Compatibility Definition Document. Dieses Dokument beinhaltet alle Eckdaten, die ein Android-Gerät sowohl softwaretechnisch als auch hardwaretechnisch erfüllen muss, um als Android-Gerät akzeptiert zu werden. Da das System grundsätzlich offen implementiert ist, stellt dieses Dokument sozusagen eine Vereinbarung dar, an die sich ein Hardwarehersteller halten sollte, um ein Android-Gerät zu entwickeln. Durch die Vereinbarung wird sichergestellt, dass:

1. das Android SDK in der unterstützten Version ohne Änderung läuft
2. sich das System den SDK-Spezifikationen gemäß verhält, auch wenn optionale Hardwarekomponenten nicht verfügbar sind
3. sich Entwickler darauf verlassen können, dass API-Funktionen genau das machen, was sie sollen, und nicht umgedeutet sind

Durch das CDD kann man sich ein gutes Bild davon machen, wie ein Android-Gerät mindestens aussieht, was es haben **muss**, was es haben **sollte** und was es haben **könnte**.

Die aktuelle Definition bezieht sich auf Android 2.3.

Im CDD werden die Worte MUST, SHOULD und MAY verwendet. In englischsprachigen Definitionen sind diese Ausdrücke folgendermaßen zu werten:

MUST	Muss	Die Funktion/Komponente muss implementiert werden. Ohne geht es gar nicht.
SHOULD	Sollte	Die Funktion/Komponente sollte implementiert werden, da ohne die Funktion/Komponente wichtige Funktionen nicht realisiert werden können. Eine sinnige Auffassung wäre: Es wäre sinnvoll.
MAY	Kann	Die Funktion/Komponente kann implementiert werden, wenn die Funktion/Komponente nicht implementiert wird, macht das aber auch nichts. Wichtige Funktionen sind dadurch nicht betroffen.
SHOULD NOT	Sollte nicht	Das kann auch vorkommen. Dabei geht es darum dass eine Funktion/Komponente zwar implementiert werden kann (MAY), aber eigentlich nicht implementiert werden sollte. Das können Funktionen/Komponenten sein, die vielleicht mal als sinnvoll erachtet waren, aber eigentlich nicht benötigt werden.

Tabelle 1.2: **Lesart der Worte MUST, SHOULD, MAY, SHOULD NOT**

1.5.1 Bildschirm

Der Bildschirm eines Android-Geräts kann theoretisch jede Dimension annehmen, so lange folgende Eckdaten eingehalten werden:

1. Eine Bildschirmdiagonale von mindestens 2.5 Zoll (6,35 cm)
2. Punktdichte mindestens 100 dpi
3. Das Seitenverhältnis (lange Seite zur kurzen Seite) muss zwischen 4:3 und 16:9 liegen. Bei den Tablet-Geräten (Android 3.0) wird hier noch das Verhältnis 16:10 dazu kommen.
4. Jedes Pixel muss quadratisch sein (ein Pixel ist kein Quadrat, korrekt muss es heißen der Abstand der Pixel in vertikaler und horizontaler Richtung muss gleich sein …[10])

GERÄT	W	H		ZOLL	PIXEL	DPI	ASPEKT RATIO		DENSITY	SIZE
Motorola Xoom	1280	800	WXGA	10,10	1509	149	1,6	16:10	mdpi	xlarge
HTC Desire	800	480	WVGA	3,7	933	252	1,7	5:3	hdpi	normal
Samsung Galaxy Tab	1024	600	WSVGA	7	1187	170	1,7	5:3	mdpi	large
HTC Dream	480	320	HVGA	3,2	577	180	1,5	3:2	mdpi	normal
Samsung Galaxy S I9000	800	480	WVGA	4	933	233	1,7	5:3	hdpi	normal
Dell Streak	800	480	WVGA	5	933	187	1,7	5:3	mdpi	large
Motorola Droid X	854	480	FWVGA	4,3	980	228	1,8	16:9	hdpi	normal

Tabelle 1.3: **Bildschirmabmessungen einiger Android-Geräte**

Bei der Programmierung für unterschiedliche Bildschirmtypen sollten nie die Abmessungen in Pixel benutzt werden. Android abstrahiert die Abmessung und die Auflösung mit allgemeinen Klassifikationen, die hier in der Tabelle in den letzten beiden Spalten stehen.

In der Tabelle ist auch zu erkennen, dass das HTC Dream mit einem Seitenverhältnis von 3:2 nicht mehr der Android 2.3-Spezifikation entspricht und das Motorola Xoom von der 2.3-Spezifikation ebenfalls nicht abgedeckt würde. Hier greift dann die Spezifikation für Honeycomb.

Hier kann man sich schon mal ein bisschen mit den Begriffen vertraut machen, wenn im weiteren Verlauf von Bildschirmausflösungen gesprochen wird. In der Spalte **Zoll** finden wir die Bildschirmdiagonale in Zoll. Die Punktdichte des Bildschirms können wir nun ermitteln,

10 A Pixel Is Not A Little Square, Technical Memo 6, Alvy Ray Smith, 17. Juli 1995, http://alvyray.com/memos/6_pixel.pdf (abgerufen 12. März 2011, 18:00)

indem wir die Anzahl der Pixel auf der Diagonalen durch die Länge der Diagonale teilen. Die Anzahl der Pixel auf der Diagonalen erhalten wir vom guten alten Pythagoras: **<equation>** $c^2 = a^2 + b^2$ **</equation><equation>** => $c = sqr\,(a^2 + b^2)$**</equation>**. Das Seitenverhältnis erhalten wir durch **<equation>** $a = w/h$ **</equation>**.

Wichtig werden die Zusammenhänge, wenn es um die Darstellung von Icons, Bitmaps oder anderen grafischen Elementen geht. Es ist leicht zu erkennen, dass die Abstände zwischen den Bildpunkten auf dem Display, bei unterschiedlichen Auflösungen und unterschiedlichen Punktdichten, unterschiedlich groß sind. Ein Ball, den wir mit einem Durchmesser von 160 Pixeln auf den Bildschirm zeichnen, hat auf einem Bildschirm mit 160 dpi (mdpi) einen Durchmesser von 2,54 cm, denn: 1 Zoll = 2,54 cm, 160 Pixel Durchmesser entsprechen bei 160 Pixeln/Zoll einem Zoll. Auf einem Bildschirm mit 240 dpi (hdpi) entspräche ein Ball mit 160 Pixeln Durchmesser allerdings: 160 Pixel * (1 Zoll/240 Pixel) = 0,67 Zoll = 1,7 cm. Der Ball ist dann auf dem hochauflösenden Bildschirm um den Faktor 1,5 kleiner.

Wenn wir die Größe von Elementen auf dem Bildschirm angeben, stellt uns Android verschiedene Maßeinheiten zur Verfügung. Dabei wird zwischen geräteabhängigen Maßangaben und geräteunabhängigen Maßangaben unterschieden.

MASSEINHEIT	BESCHREIBUNG
dp	Von der Auflösung unabhängige Bildpunkte (Density-independent Pixels). Diese Einheit verhält sich relativ zu 160 dpi. 160 dp sind immer ein Zoll, egal auf welchem Bildschirm. Diese Maßeinheit wird hauptsächlich für Maßangaben in Layouts verwendet. Diese Maßeinheit ist geräteunabhängig.
sp	Von der Skalierung unabhängige Pixel (Scale-independent Pixel). Diese Maßangaben verhalten sich wie die dp-Maßangaben, berücksichtigen aber den Schriftskalierungsfaktor, den der Benutzer eingestellt hat. Wird hauptsächlich für Schriftgrößen verwendet, sodass die aktuelle Bildschirmauflösung und die Benutzereinstellungen berücksichtigt werden. Diese Maßeinheit ist geräteunabhängig.
pt	Point, 1/72 eines Zolls. Diese Maßeinheit ist geräteunabhängig.
px	Pixel. Ein Pixel korrespondiert mit einem Bildpunkt des Bildschirms. Diese Maßeinheit ist **geräteabhängig** und deshalb nicht zur Verwendung empfohlen.
mm	Millimeter. Diese Maßeinheit ist geräteunabhängig.
in	Inches (Zoll). Diese Maßeinheit ist geräteunabhängig.

Tabelle 1.4: **Maßeinheiten**

Wir werden sehen, dass wir innerhalb von Layout- und Style-Ressourcen die Maßeinheiten bei Größenangaben u.Ä. benutzen oder aber innerhalb von Dimensionsressourcen einzelne Werte mit Maßeinheiten zur späteren Verwendung definieren können. Wenn wir allerdings selbst auf dem Bildschirm zeichnen, benutzen wir bei Aufruf der Grafikprimitiven (Linie, Rechteck etc.) die Einheit Pixel. Das bedeutet, dass wir uns um eine geräteunabhängige Umrechnung unserer Maße selbst kümmern müssen, z.B. indem wir uns auf die Dots per Inch des Bildschirms beziehen.

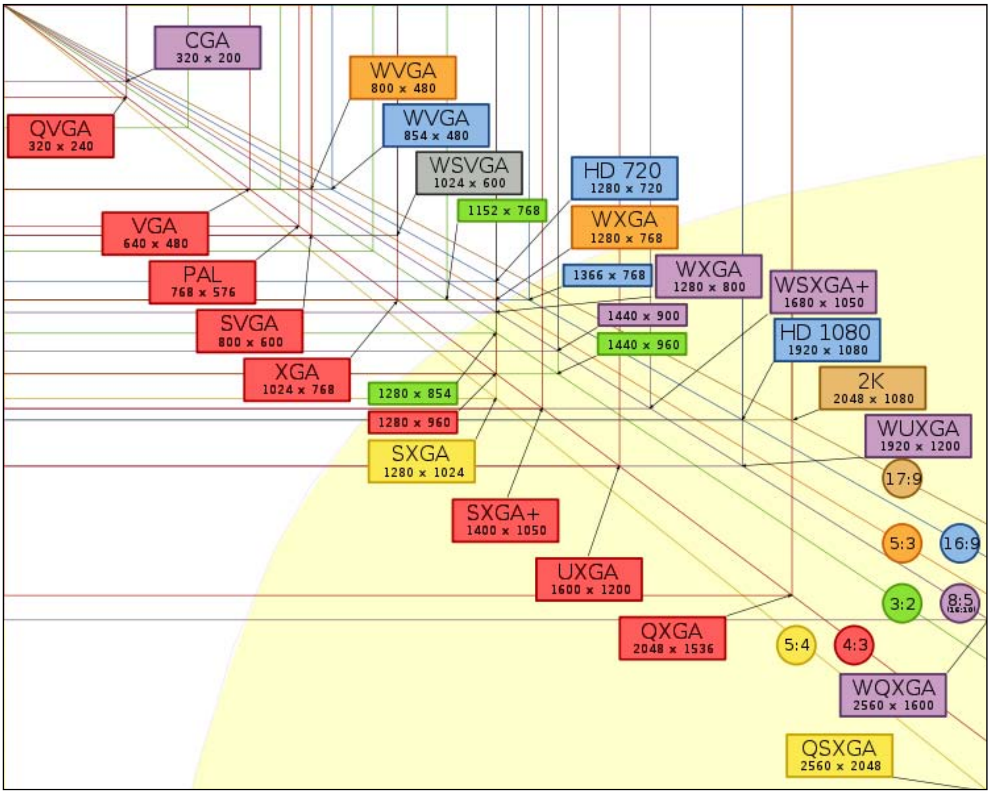


Abbildung 1.1: **Übliche Bildschirmauflösungen und ihre Bezeichnung**[11]

Natürlich ist auch leicht erkennbar, dass die unterschiedlichen Abmessungen des Bildschirms selbst Einfluss auf die Gestaltung der Anwendung haben. Auf einem Tablet wie dem Motorola Xoom haben wir viel mehr Platz zur Verfügung als auf dem Display eines HTC Desire, und man würde das Layout einer Applikation für das Tablet anders wählen als für das Smartphone. Genau wie beim API-Level soll man seiner Applikation mitgeben, welche Bildschirmgrößen die Applikation unterstützt. Nach diesem Kriterium werden die Applikationen dann im Market entsprechend gefiltert.

11 http://en.wikipedia.org/wiki/Graphic_display_resolutions (abgerufen 5. Februar 2011, 09:00 MEZ)

Wenn wir uns später mit der Unterstützung unterschiedlicher Geräte beschäftigen, werden wir uns anschauen, wie man Applikationen geschickt für den Betrieb auf unterschiedlichen Geräten vorbereiten kann.

Neben der eigentlichen Auflösung spielt auch die Ausrichtung (Orientation) des Bildschirms eine Rolle.

Der Bildschirm kann bei den Android-Geräten hochkant, d.h. im Porträtmodus, oder quer, das heißt im Landscapemodus, betrieben werden.

Android-Geräte sind so konzipiert, dass das Gerät eine natürliche Ausrichtung hat, in der das Betriebssystem startet. Das HTC Desire und ähnliche Geräte haben den Porträtmodus als natürliche Ausrichtung. Andere Geräte, die z.B. eine querformatige Tastatur ausklappen können, arbeiten ggf. standardmäßig im Landscape-Modus.

INFO

Es gibt keine direkte Methode um zu ermitteln, welches die natürliche Ausrichtung des Geräts ist. Mittels `getWindowManager().getDefaultDisplay().getRotation()` **können wir aber herausfinden, ob das Gerät gegenüber seiner natürlichen Ausrichtung um 90, 180 oder 270° gedreht wurde.**

Wenn wir nun per `getWindowManager().getDefaultDisplay().getWidth()` **und** `getWindowManager().getDefaultDisplay().getHeight()` **die Breite und Höhe vergleichen, so wissen wir: Ist die Höhe größer als die Breite und das Display ist nicht gedreht (bzw. 0° oder 180°), haben wir ein Gerät im Hochkantformat, ansonsten im Querformat. Ist die Breite größer als die Höhe und das Display ist nicht gedreht, haben wir ein Gerät im Querformat, ansonsten im Hochformat.**

Tablets werden in der Regel das Querformat als natürliche Ausrichtung haben[12].

Android-Anwendungen können so konzipiert werden, dass sie entweder einen entsprechenden Modus anfordern oder dynamisch auf die Änderung der Ausrichtung reagieren. Zu diesem Zweck muss ein Android-Gerät immer Auskunft über die aktuelle Ausrichtung des Geräts geben können. Wie wir später sehen werden, übernimmt das Betriebssystem viele Aufgaben beim Drehen des Geräts automatisch, wir werden aber auch sehen, wie wir das abstellen und selber darauf reagieren können.

Das Bildschirmkoordinatensystem selbst ist zweidimensional. In der natürlichen Ausrichtung zeigt die y-Achse nach unten, die x-Achse nach rechts. Der Ursprung (0,0) des Koordinatensystems liegt in der linken oberen Ecke. Wird das Gerät nun gedreht, wird das Koordinatensystem des Bildschirms so angepasst dass die y-Achse wieder nach unten und die x-Achse nach rechts zeigt. Daher müssen wir beim Zeichnen auf dem Bildschirm im Prinzip auch nichts weiter berücksichtigen, außer dass sich Breite und Höhe ändern. Indem wir unsere Layouts entsprechend dynamisch aufbauen, müssen wir uns bei den Benutzeroberflächen fast gar keine Gedanken um die Rotation machen.

12 Logisch. Wer legt sein Frühstücksbrett schon hochkant vor sich hin ;-)

ACHTUNG

Durch das Drehen eines Geräts wird, wenn es nicht durch die Konfiguration der Activity unterbunden wird, das Koordinatensystem so angepasst, dass die y-Achse immer nach unten zeigt. Das Koordinatensystem der Sensoren wird beim Drehen aber nicht angepasst. Was das für Auswirkungen hat und wie man damit umgeht, betrachten wir beim Thema Sensoren.

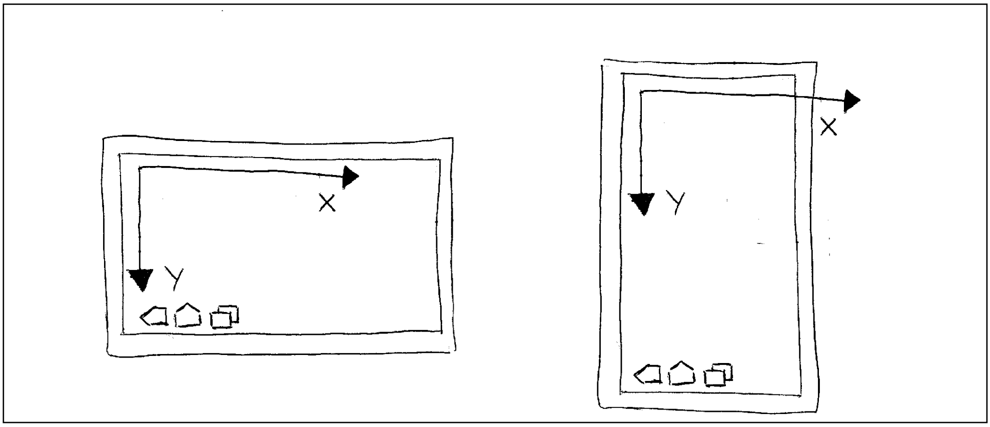


Abbildung 1.2: **Drehen eines Tabletts nach links**

INFO

Folgende Stichwörter sind wichtig im Zusammenhang mit dem Bildschirm:

1. **Display.getMetrics(DisplayMatrics);**
2. **getWindowManager().getDefaultDisplay();//Innerhalb einer Activity**
3. **getWindowManager().getDefaultDisplay().getOrientation(); //Innerhalb einer Activity**
4. **getResources().getConfiguration(); //Innerhalb einer Activity**
5. **Configuration.screenLayout, Configuration.uiMode**
6. **<supports-screens /> im AndroidManifest.xml-File**

1.5.2 Eingabegeräte

Tastatur

Jedes Android-Gerät muss mindestens eine virtuelle Tastatur bereitstellen, unabhängig davon, ob das Gerät eine echte Tastatur besitzt. Das ist insofern auch vernünftig, da der Trend dazu geht, Tastaturen per Bluetooth oder Docking-Station an Geräte anzudocken, die ansonsten hauptsächlich per Touchscreen bedient werden. Android ist so aufgebaut, dass weitere Eingabetreiber realisiert werden können, um z.B. alternative virtuelle Tastaturen zur Verfügung zu stellen.

Wenn ein Gerät eine Tastatur hat muss es entweder eine vollwertige QWERT…- oder eine Zwölf-Tasten-Tastatur sein. Die Zwölf-Tasten Tastatur entspricht dabei der üblichen Telefontastatur.

»Non-Touch«-Navigation

Viele Geräte bieten zur Navigation, ähnlich der Cursor-Tasten und/oder der Maus bei einem normalen PC/Laptop, einen Trackball, ein D-Pad (Directional Pad) oder ein Dreh-/Scrollrad an.

Allerdings muss ein Gerät keine eigenständige Navigation anbieten, dafür müssen aber Alternativen über den Touchscreen implementiert sein.

Ein Trackball kann tatsächlich als »Ball« ausgeführt sein, bei einigen Geräten kommt jedoch ein kleiner optischer Trackball zum Einsatz.

Ein D-Pad ist eine Schaltwippe mit vier Richtungen und einem Mitteldruckpunkt, die man z.B. von Spielekonsolen her kennt.

Ein Scrollrad ist üblicherweise so angeordnet, dass es bequem mit dem Daumen zu bedienen ist, und dient hauptsächlich zum Durchblättern von Listeneinträgen oder dem Durchführen eines vertikalen Bildlaufs (vertikales Scrolling).

Wie alle anderen Hardwareanforderungen auch kann jede Applikation bestimmen was für ein Eingabegerät benötigt wird. Wichtig ist dabei zu wissen, dass in der Regel D-Pad-Ereignisse auch in Trackball-Ereignisse bzw. umgekehrt gewandelt werden, auch Scrollrad-Ereignisse landen als Trackball- und D-Pad-Ereignisse beim Gerät. Allerdings sollte man sich entscheiden, wie man die Ereignisse behandelt, denn ein Trackball liefert zwar genauere Positionsangaben als ein D-Pad, ein D-Pad ist dafür ggf. leichter und genauer zu bedienen.

Daher ist es empfehlenswert, sich in der Applikation einen Überblick darüber zu verschaffen was das Gerät eigentlich zu bieten hat.

INFO

Folgende Stichwörter sind wichtig im Zusammenhang mit den Eingabegeräten:

1. **View.onKeyDown(KeyEvent), View.onKeyUp(KeyEvent)**
2. **View.onTrackballEvent(MotionEvent)**
3. **getResources().getConfiguration(); //Innerhalb einer Activity**
4. **Configuration.keyboard, Configuration.navigation**
5. **<uses-configuration /> im AndroidManifest.xml – File**

Weitere Navigationstasten

Startseite (Home), Menü (Menu) und Zurück (Back) sind essenzielle Funktionen im Android-Navigationsschema. Daher müssen Android-Geräte diese drei Tasten bereitstellen, wobei diese nicht unbedingt als Hardwaretasten ausgeführt sein müssen. Die Funktionen müssen allerdings immer verfügbar sein.

INFO

Bei den meisten Smartphones sind diese Tasten als Hardwaretasten ausgeführt. Für Tablets wurde in Android 3 die *System Bar* eingeführt. Die *System Bar* bietet zum einen die o.g. Tasten als virtuelle Tasten an, zum anderen werden in der *System Bar* auch Statusmeldungen verwaltet.

Eine Taste zum Ausführen einer Suche sollte realisiert sein, Tasten zum Initiieren und Beenden eines Telefonanrufs können angeboten werden.

Auf die Home-, Menu- und Back-Tasten reagiert man in seiner Anwendung in der Regel nicht direkt, diese Funktionalität wird durch die Laufzeitumgebung definiert. Die Laufzeitumgebung kontrolliert diese Tasten und löst entsprechende Funktionen aus. Durch Betätigen der Back-Taste geht man im Aktivitätenstapel (Activity-Stack) eins zurück, Home katapultiert uns auf den Startbildschirm des Geräts, und Menu ruft das Auswahlmenü der Aktivität auf.

Die Search-Taste wiederum startet das Search-Framework, das eine konsistente Suchstrategie über das gesamte System bietet. Wir müssen keine eigene Suchboxen oder Suchdialoge erstellen, sondern können unsere Applikation in das Search-Framework einklinken.

Touchscreen

Touchscreens sind obligatorisch. Jedes Android-Gerät muss einen Touchscreen bereitstellen, dabei kann es sich um einen kapazitiven oder einen resistiven Touchscreen handeln.

Ganz grob erklärt funktionieren die kapazitiven Touchscreens folgendermaßen: Die Oberfläche des Bildschirms wird mit einer leitfähigen Schicht versehen, an die ein gleichmäßiges elektrisches Feld angelegt wird. Die Oberfläche ist gleichmäßig aufgeladen. Nähert sich der Finger der Oberfläche, springt ein Teil der Ladung auf den Finger über, diese Störung (die Änderung der Ladung der Oberfläche) wird gemessen und in eine entsprechende Position umgewandelt.

Kapazitive Touchscreens lassen sich aufgrund ihrer Funktionsweise nur mit bloßen Fingern oder leitfähigen Stiften bedienen. Das stellt im Winter (Handschuhe) und vor allem für Menschen mit Handprothesen ein unüberwindliches Hindernis dar.

Allerdings bieten die kapazitiven Touchscreens einige Vorteile. So kann man mit dieser Technologie recht einfach Multitouch-Erkennung realisieren (also mehrere Punkte/Bewegungen gleichzeitig auswerten), und der Einfluss auf die Bildschirmhelligkeit ist geringer

Resistive Touchscreens sind mit einem Gitter von Drähten (vereinfacht ausgedrückt) ausgestattet, wobei sich die Drähte nicht berühren. Zwischen den vertikalen Drähten und den horizontalen Drähten befindet sich eine Lücke. An die Drähte wird ein Gleichstrom angelegt. Drückt man nun auf dieses Gitter, berühren sich die vertikalen und die horizontalen Drähte

in einem Kreuzungspunkt, und der Widerstand innerhalb des Leiters ändert sich. Diese Änderung des Widerstands wird gemessen und in die Position umgerechnet. Der Vorteil ist, dass die Messung sehr genau ist (die Gitter sind sehr fein), und da die Bauart auf Druck reagiert, kann man diese Touchscreens mit allem Möglichen bedienen, mit einem Stift bzw. Stylus, vom Finger über den Fingernagel bis hin zum Holzstöckchen. Allerdings ist die Bauweise komplexer (wenn auch zurzeit wohl noch günstiger) als bei kapazitiven Touchscreens und die Auswertung von Multitouch-Gesten aufwendiger.

Das Schöne ist, dass Android diese Details vor uns verbirgt, das System verhält sich unabhängig von der verwendeten Technologie in weiten Teilen konsistent. Allerdings gibt es je nach Bauart einige Gesten, die nicht erkannt werden oder nur schwer umzusetzen sind.

Der kapazitive Touchscreen hat wie beschrieben bauartbedingt den Vorteil, dass kein Druck ausgeübt werden muss, um die Berührung zu messen. So kann dann auch Bewegung über die Oberfläche recht gut erkannt werden, vor allem muss der Benutzer keinen Druck dabei ausüben. Damit sind Wisch- bzw. Schleuderbewegungen (Flings) sehr anschmiegsam zu realisieren. Ein großer Teil der Faszination der Smartphone-Oberflächen macht ja gerade die Bedienung über Wischen und Schleudern aus, vor allem in Verbindung mit dem kinetischen Rollen, das abhängig von der Schleudergeschwindigkeit gesteuert wird und sogar noch einen sogenannten »Overshoot« hinlegt, also über das Ende hinausschießt und elastisch wieder zurückkommt.

Android bietet innerhalb des Frameworks einen ganzen Strauß von Möglichkeiten an, auf Touchscreen-Ereignisse zu reagieren. Einfache Ereignisse wie das Berühren des Bildschirms können direkt in den Sichten (Views) behandelt werden, für komplexere Bewegungen gibt es sogenannte Gesture-Detektoren, die in letzter Konsequenz sogar den Aufbau eigener Gestenalphabete erlauben.

INFO

Folgende Stichwörter sind wichtig im Zusammenhang mit Touchscreens:

1. **View.onTouchEvent(MotionEvent)**
2. **GestureDetector**
3. **GestureDetector.OnGestureListener**
4. **GestureDetector.OnDoubleTapListener**
5. **GestureOverlayView**
6. **GestureLibrary**
7. **GestureUtils**
8. **getResources().getConfiguration(); //Innerhalb einer Activity**
9. **Configuration.touchscreen**
10. **<uses-configuration /> im AndroidManifest.xml – File**

1.5.3 Sensoren

Neben den Möglichkeiten die die Eingabe und Steuerung per Touchscreen bieten, finde ich die unterschiedlichen Sensoren, die ein Smartphone heute besitzt, ungeheuer spannend und reizvoll.

Durch die Sensoren werden erst einige Funktionen möglich, die den Reiz mobiler, vernetzter Geräte ausmachen. Im Abschnitt über Bildschirmformate haben wir bereits kurz angesprochen, dass die Smartphones in unterschiedlicher Orientierung (Porträt, Landscape) benutzt werden können. Damit das Betriebssystem die Orientierung des Bildschirms erkennen kann braucht das Gerät Sensoren, die die Orientierung des Geräts im Raum ermitteln. Kann man aber erst einmal die Orientierung des Geräts ermitteln, lassen sich darüber auch neuartige Steuerungen in einem Spiel realisieren.

Um die Position des Geräts auf diesem Planeten herauszufinden, benötigt das Gerät irgendeinen Sensor, der die Position ermitteln kann. Der beste Sensor für diesen Zweck ist ein GPS-Empfänger. Wir werden im Folgenden aber noch sehen, dass das nicht die einzige Möglichkeit für eine Positionsbestimmung ist, es ist aber die Möglichkeit mit der besten Genauigkeit. Aber warum will man überhaupt die Position ermitteln? Das Zauberwort ist *Location Based Services*, ortsbezogene Dienste. Die einfachste Form ist die klassische Navigation von A nach B (möglicherweise noch über C). Dann möchte man noch wissen: *Was ist in der Nähe?* (*Points of Interest, POIs*), und dazu noch genauere Informationen erhalten. Wenn man das Ganze weiterdenkt, und das haben die bekannten sozialen Netzwerke ja bereits getan, dann ist es ein kleiner Schritt, anderen zu sagen *Hier bin ich* (z.B. *Facebook places*) oder auch zu fragen *Wo bist Du gerade?*. Nimmt man nun die Möglichkeit hinzu, die Orientierung des Geräts im Raum zu ermitteln dann kann man auch noch die Frage stellen: *Was sehe ich gerade?* (*Augmented Reality*). Bei ortsbezogenen Diensten ist der Fantasie keine Grenze gesetzt. Es wäre denkbar, sich an die Einkaufsliste erinnern zu lassen, wenn man in die Nähe eines Geschäfts kommt oder sich nachts den Sternenhimmel erklären zu lassen.

Eine schöne Form ortsbezogener Anwendungen ist, finde ich, das Geocaching[13], und Spiele wie die altehrwürdige Schnitzeljagd erleben durch die GPS-Fähigkeiten der Geräte eine moderne Renaissance.

Es verwundert daher kaum, dass Google zusätzliche Klassen für die Nutzung von Google Maps in Verbindung mit Android zur Verfügung stellt.

Was mich persönlich sehr begeistert hat, ist, dass ich mich auch mathematisch und physikalisch anstrengen musste, denn um die Sensoren zu verstehen und die Sensorwerte zu nutzen, muss man sich mit Koordinaten, Vektoren, Beschleunigungen, Kreisgeschwindigkeiten, Filtern und verschiedenen Koordinatensystemen auseinandersetzen.

13 http://de.wikipedia.org/wiki/Geocaching (abgerufen 5. Februar 2011, 09:00 MEZ)

Koordinatensysteme

Im Abschnitt über den Bildschirm haben wir bereits erfahren, dass ein Gerät eine natürliche Ausrichtung besitzt. Im Grunde kann man das immer daran erkennen, wo die Tasten oder das Mikrofon und Lautsprecher angeordnet sind. Bei Geräten, die keine Tasten besitzen – und das wird bei den meisten Tablets der Fall sein –, erkennt man die Ausrichtung daran, wo die Kamera oder das Logo des Herstellers platziert sind. Tablets werden in der Regel als natürliche Ausrichtung das Querformat haben.

Es gibt nun unterschiedliche Koordinatensysteme, mit denen wir uns auseinandersetzen müssen.

Da ist als erstes das Koordinatensystem des Bildschirms. In der natürlichen Ausrichtung zeigt die y-Achse nach unten und die x-Achse nach rechts. Wird das Gerät nun gedreht, wird das Koordinatensystem des Bildschirms so angepasst, dass die y-Achse wieder nach unten und die x-Achse nach rechts zeigt, es sei denn wir vermeiden die automatische Anpassung des Bildschirms an die Lage des Geräts.

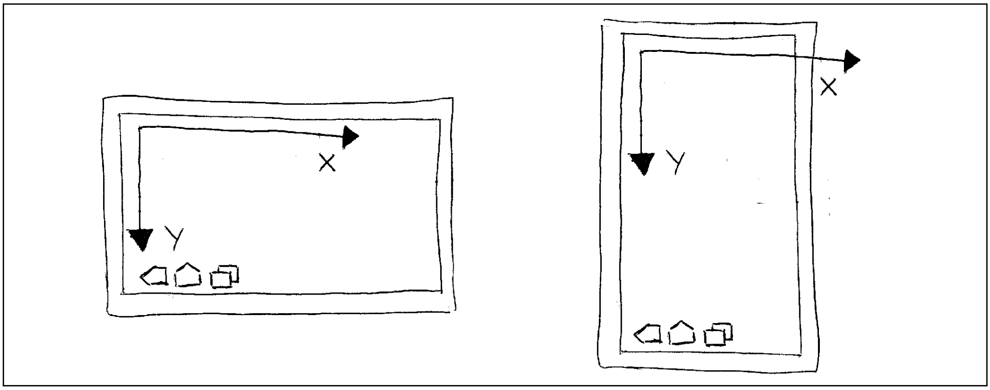


Abbildung 1.3: **Koordinatensystem Bildschirm**

Als Nächstes gibt es das Koordinatensystem der Sensoren.

Das Sensorkoordinatensystem stimmt in der Achsenausrichtung mit dem Bildschirmkoordinatensystem überein, **solange das Gerät in der natürlichen Ausrichtung gehalten wird**. Die y-Achse zeigt allerdings **nach oben**, die x-Achse nach rechts, und die z-Achse zeigt aus dem Display heraus auf uns (genauer gesagt senkrecht aus der Frontseite des Bildschirms heraus). Der Ursprung des Koordinatensystems liegt genau im Zentrum des Geräts. Die Achsen des Sensorkoordinatensystems werden aber **nicht** vertauscht, sobald das Gerät gedreht oder gekippt wird.

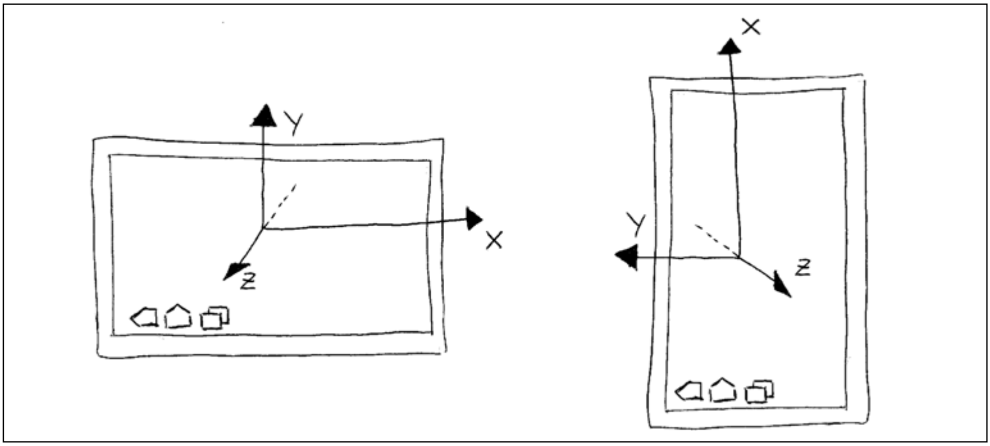


Abbildung 1.4: **Koordinatensystem Sensoren (Querformat ist natürliche Lage)**

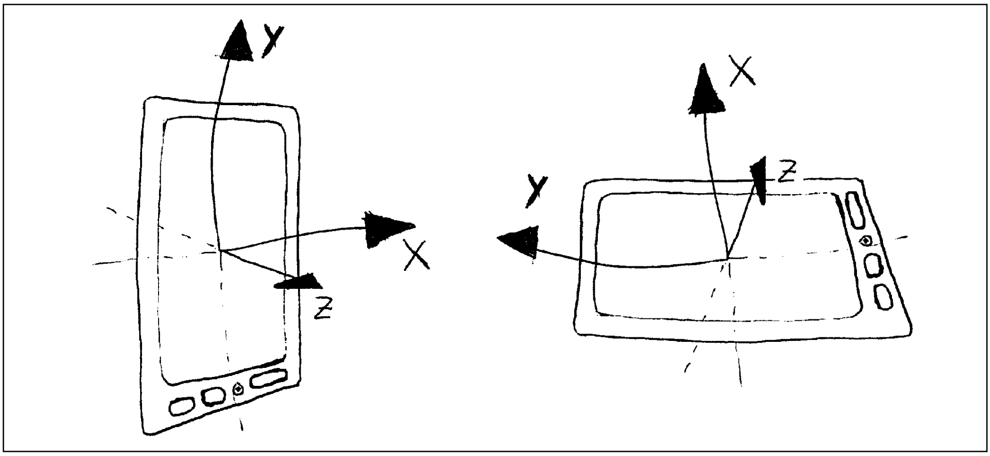


Abbildung 1.5: **Koordinatensystem Sensoren (Hochformat ist natürliche Lage)**

Ein weiteres Koordinatensystem ist das Weltkoordinatensystem, das als Bezug für die Ermittlung des Rotationsvektors und der Inklination[14] des Geräts dient. Im Weltkoordinatensystem zeigt die y-Achse tangential zur Erdkrümmung in Richtung des magnetischen Nordpols, die x-Achse tangential zur Erdkrümmung in Richtung Osten und die z-Achse aus dem Erdmittelpunkt heraus senkrecht Richtung Himmel.

14 Die Inklination ist in diesem Zusammenhang die Neigung der Feldlinie des Erdmagnetfeldes zur Horizontalen. Am Äquator beträgt die Inklination 0°, an den Polen 90°, in Deutschland zwischen 67° und 70°. Bildlich ist dies das Maß der Neigung einer Kompassnadel gegenüber der Horizontalen. Je näher man dem magnetischen Pol kommt, umso stärker kippt die Nadel in Richtung Boden.

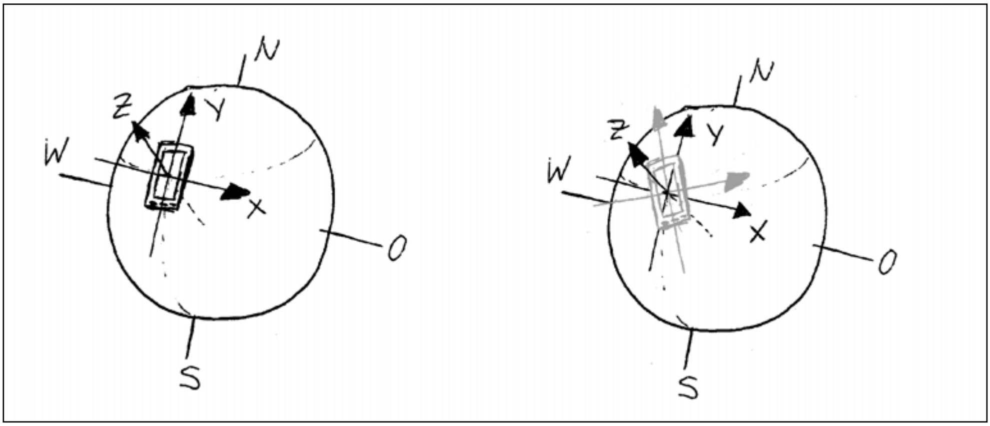


Abbildung 1.6: **Welt- und Sensorkoordinaten, Rotation**

Wenn wir nun also das Gerät so in natürlicher Ausrichtung auf den Tisch legen, dass die y-Achse des Bildschirms (**Achtung, die y-Achse zeigt nach unten!)** genau nach Süden – das heißt die Oberkante des Bildschirms zeigt nach Norden – und die x-Achse nach Osten zeigt, dann stimmen alle drei Koordinatensysteme so ziemlich miteinander überein. Ein Rotationssensor sollte dann auch für alle Komponenten einen Winkel von 0° auswerfen, ein Beschleunigungssensor (der die Erdbeschleunigung G nicht kompensiert) sollte in z-Richtung eine Beschleunigung von 9,81 m/s² liefern. (Die »Erdanziehung« zerrt in negative z-Richtung an unserer z-Achse, die z-Achse wird also mit 9,81 m/s Richtung Boden beschleunigt).

Die Ausrichtung des Geräts ist also immer quasi der Winkel einer Drehung unseres Gerätekoordinatensystems um die Achsen des Weltkoordinatensystems. Und je nachdem, wie wir das Gerät in der Welt ausrichten, haben das Magnetfeld und das Gravitationsfeld unterschiedliche Einflüsse auf die Sensorkomponenten (Beschleunigungssensor, Gravitationssensor, Magnetfeldsensor), und daraus lässt sich wiederum die Rotationsmatrix berechnen.

Warum muss man aber nun Acht geben, wenn zwar das Bildschirmkoordinatensystem beim Drehen angepasst wird, das Sensorkoordinatensystem jedoch nicht?

Am einfachsten kann man sich das anhand eines Pfeils erklären, den wir zeichnen wollen, der immer nach unten zeigen soll, wenn wir das Gerät vor uns halten und drehen. Nehmen wir eine einfache Implementierung, die die Beschleunigungsachsen auswertet und einfach Pfeile mit einer der Beschleunigung entsprechenden Länge zeichnet.

Halten wir das Gerät ruhig in natürlicher Ausrichtung (Sensorkoordinatensystem und Bildschirmkoordinatensystem stimmen noch überein), dann liefert der Beschleunigungssensor in Y-Richtung rund 9,81 m/s², in X-Richtung nahe 0 m/s². Wenn wir also einen Pfeil in Y-Richtung mit einer entsprechenden Länge zeichnen, dann zeigt uns dieser Pfeil die Richtung zum Boden an. Drehen wir nun das Gerät in die andere Ausrichtung, passt Android das Bildschirmkoordinatensystem an, das Sensorkoordinatensystem jedoch nicht! Je nachdem, wie wir gedreht haben, zeigt jetzt die x-Achse des Sensors nach unten (oder nach oben ...) und liefert einen Beschleunigungswert von +/-9,81 m/s² (je nach Drehrichtung), die y-Achse

liefert nahe 0 m/s². Die y-Achse des Bildschirms jedoch zeigt immer noch hartnäckig nach unten. Wenn wir jetzt den Y-Pfeil zeichnen, wird dieser aber die 0 m/s² darstellen und der X-Pfeil die +/-9,81 m/s², und unser »Lot« zeigt nach rechts oder links. Es ist offensichtlich, dass wir also in diesem Fall die Drehung des Geräts berücksichtigen müssen, um die Sensorwerte entsprechend zu tauschen.

Ein anderes Beispiel ist ein Murmelspiel, bei dem die Murmel in die Richtung rollt, in die wir das Gerät kippen, **und** dabei noch die in die entsprechende Richtung wirkende Beschleunigung erfährt.

INFO

Immer wenn wir die Orientierung des Geräts im Raum oder den Einfluss der Beschleunigungssensoren als Steuerungselement benutzen wollen, die Werte also einen Einfluss auf das Bildschirmkoordinatensystem haben, müssen wir

a) die natürliche Lage des Geräts und

b) die Drehung des Geräts gegenüber der natürlichen Lage berücksichtigen

Auch wenn wir, z.B. bei einem Spiel, die Ausrichtung des Bildschirms auf Querformat fixieren, müssen wir immer noch berücksichtigen, wie das Gerät gegenüber der natürlichen Lage ausgerichtet ist. Je nach natürlicher Ausrichtung ist das Sensorkoordinatensystem unterschiedlich ausgerichtet.

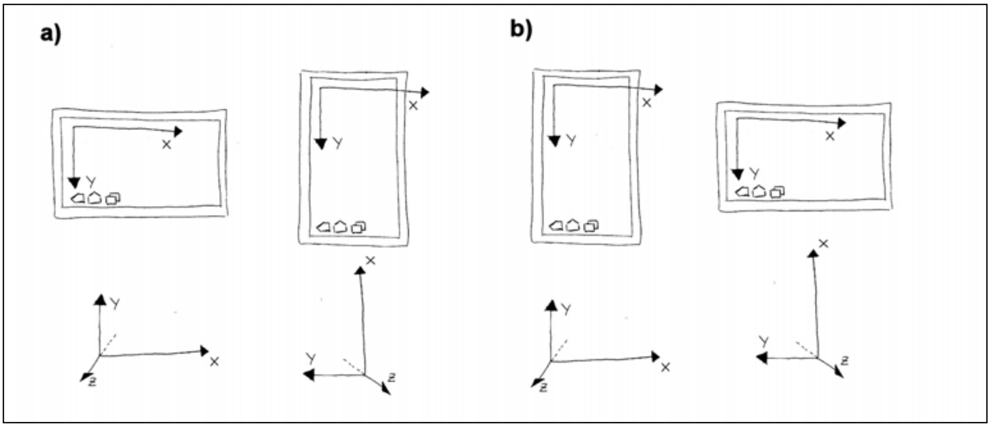


Abbildung 1.7: **Vergleich Bildschirmkoordinaten zu Sensorkoordinaten**

Die Abbildung verdeutlicht das. Fall **a)** ist ein Gerät, das seine natürliche Ausrichtung im Querformat hat. Wollen wir hier den Einfluss des Beschleunigungssensors für die Bewegung einer Kugel auf dem Bildschirm einsetzen, können wir die Werte in X-Richtung und in Y-Richtung von den X-Werten und Y-Werten des Sensors direkt übernehmen.

Im Fall **b)** müssen wir stattdessen im Querformat für die Bewegung in Y-Richtung den X-Wert des Sensors auslesen und für die Bewegung in X-Richtung den Y-Wert des Sensors.

Ein wesentlich komplexerer Fall tritt ein, wenn wir die Orientierung des Geräts im Raum dazu verwenden wollen, die Himmelsrichtung bzw. den Azimut und möglicherweise noch

den Blickwinkel zwischen Horizont und Zenit zu ermitteln. Das ist z.B. nötig, wenn wir Anwendungen für Augmented Reality entwickeln und in unser Kamerabild standort- und himmelsrichtungsabhängige Informationen einblenden wollen.

In diesem Fall kommt dann nämlich noch hinzu, dass das Gerätekoordinatensystem gegenüber dem Weltkoordinatensystem gekippt ist. Wenn wir also die Drehung des Geräts um die z-Achse der Weltkoordinaten ermitteln wollen (und damit die Himmelsrichtung), ist es natürlich entscheidend ob das Gerät wie ein Kompass benutzt wird (Sensor-z-Achse zeigt in Richtung Welt-z-Achse) oder wie eine Kamera gehalten wird (Sensor-y-Achse zeigt entgegengesetzt in Richtung Welt-z-Achse). Im ersteren Fall ist alles noch ganz einfach, hier lässt sich die Rotation um die z-Achse des Sensors direkt in die Himmelsrichtung umsetzen. Beim zweiten Fall ist das aber nicht so, hier muss die Drehung um die y-Achse des Sensors als Himmelsrichtung berücksichtigt werden, da ja nicht mehr um die z-Achse, sondern um die y-Achse gedreht wird. Da die Rotation nun aber aus der Lage des Geräts im Magnetfeld und durch die Beschleunigungskomponenten ermittelt wird, muss die hier das Koordinatensystem der Rotationsmatrix angepasst werden, und zwar muss die z-Achse der Welt auf die y-Achse des Geräts abgebildet werden (das Weltkoordinatensystem wird entsprechend gekippt), und schon liefert uns die Orientierung die Himmelsrichtung als Drehung um unsere y-Achse.

Wie das genau funktioniert schauen wir uns später an einem Beispiel an.

INFO

Folgende Stichwörter sind im Zusammenhang mit den Koordinatensystemen wichtig:

1. **getWindowManager().getDefaultDisplay().getOrientation();**
2. **SensorManager.getRotationMatrix(...);**
3. **SensorManager.remapCoordinateSystem(...);**
4. **SensorManager.getOrientation(...);**
5. **SensorManager.getInclination(...);**

Einfluss auf den Strombedarf

Bei allen Sensoren ist zu beachten, dass deren Nutzung den Energiebedarf des Gesamtsystems in die Höhe treibt. Android gibt bereitwillig Auskunft über die Stromaufnahme jedes Sensors, und mittels des Battery-Managers (den man durchaus auch bei den Sensoren ansiedeln könnte) können wir auch selbst ermitteln, in welcher Geschwindigkeit sich die Batterie leert.

Es ist wichtig, dass die Sensoren nur dann aktiviert werden, wenn sie gebraucht werden, und auch nur so lange aktiv bleiben, wie man sie unbedingt benötigt.

ACHTUNG

Wenn eine Anwendung inaktiv wird, muss sie sich selbst darum kümmern, die Verwendung der Sensoren zu deaktivieren. Das Betriebssystem bzw. das Framework kümmert sich nicht darum. Wenn man das vergisst, wird die Batterie im Hintergrund leer gesaugt.

INFO

Folgende Stichwörter sind wichtig im Zusammenhang mit Sensoren:

1. getSystemService(Context.SENSOR_SERVICE);
2. SensorManager.getDefaultSensor(...);
3. SensorManager.registerListener(...);
4. SensorManager.unregisterListener(...);
5. SensorEventListener

Übermittlung der Werte

Bis Android einschließlich 2.2 liefern die Sensoren ihre Werte nur bei Wertänderung an die Anwendung aus, wobei man dem Sensor-Manager trotz allem mitteilt, wie häufig er die Werte ausliefern soll. Bei Beschleunigungssensoren, dem Magnetometer und auch anderen bemerkt man das fast gar nicht, da aufgrund von ständigen Schwankungen im Magnetfeld und der Empfindlichkeit der Sensoren auch in Ruhe ständig neue Werte anfallen (könnte man teilweise auch als Rauschen bezeichnen).

Dennoch, wenn man kontinuierliche Aufzeichnungen machen möchte, gibt das die Sensorschnittstelle eigentlich nicht her, wir erhalten aber immer einen Zeitstempel, wann die Wertänderung angefallen ist.

Ab 2.3 sind Streaming-Sensoren vorgesehen, die ihre Werte kontinuierlich abliefern können.

INFO

Folgende Stichwörter sind wichtig im Zusammenhang mit der Übermittlung von Sensorwerten:

1. SensorEventListener.onSensorChanged(...);
2. SensorEventListener.onAccuracyChanged(...);

Accelerometer (Beschleunigungssensor)

Der Beschleunigungssensor misst die Beschleunigung des Geräts in m/s^2 in Richtung der Achsen des Gerätekoordinatensystems. Technisch kann man sich einen Beschleunigungssensor als kleine Feder vorstellen, an der eine Testmasse aufgehängt ist. Durch die Auslenkung der Feder kann die Kraft bestimmt werden, die auf die Testmasse wirkt, und aus der Kraft die Beschleunigung. Nehmen wir zur Betrachtung einfach an, dass die Feder mit der Testmasse in negativer Richtung der z-Achse zeigt. Wenn das Gerät flach auf dem Tisch mit dem Display nach oben liegt, zeigt die Testmasse in Richtung Boden. Auf die Masse wirkt nun die konstante Beschleunigung $g = 9{,}81\ m/s^2$, das heißt, die Testmasse erfährt

eine konstante Beschleunigung von 9,81 m/s², das Gerät befindet sich in Ruhe und erfährt eine Beschleunigung von 0 m/s². Ohne Tischplatte würde das Gerät nun logischerweise in den freien Fall übergehen. Wenn es sich nun im freien Fall befinden würde, wirkt auf die Testmasse relativ keine Beschleunigungskraft mehr (da sich das Initialsystem nun mit einer Beschleunigung von 9,81 m/s² Richtung Zerstörung bewegt…). Der Sensor würde in Richtung der z-Achse im freien Fall 0 m/s² auswerfen.

Aber bleiben wir auf dem Tisch, das ist für das Gerät gesünder. Die obige Betrachtung erklärt aber, warum wir später in unserer Applikation eine Beschleunigung in Richtung des Bodens messen, obwohl sich das Gerät in Ruhe befindet. Daraus folgt auch gleich messerscharf, dass die eigentliche Beschleunigung des Geräts in Richtung einer Achse durch Eliminierung des Einflusses der Erdbeschleunigung auf die entsprechende Achse berechnet werden muss.

Auch in Richtung der x-Achse und in Richtung der y-Achse gibt es eine solche Feder mit einer Testmasse (das ist alles wirklich sehr schematisch erklärt!). Liegt das Gerät flach auf dem Tisch, wirken auf die Testmassen keine Beschleunigungskräfte, in Richtung der x-Achse und in Richtung der y-Achse haben wir eine Beschleunigung von 0 m/s². Kippen wir nun das Gerät über die y-Achse in Richtung einer Längsseite nimmt der Einfluss der Erdbeschleunigung auf der z-Achse kontinuierlich ab und auf der x-Achse kontinuierlich zu, bis auf der z-Achse 0 m/s² wirken und auf der x-Achse 9,81 m/s².

Somit ist leicht ersichtlich, dass über diese Art Beschleunigungssensor, der also den Einfluss der Erdbeschleunigung nicht eliminiert, die Orientierung des Geräts bereits abgeschätzt werden kann. Es ist allerdings nicht direkt möglich daraus eine Korrelation zum Azimuth[15] und zur Neigung und der Drehung in Grad herzustellen. Das kommt später.

Legen wir das Gerät wieder flach auf den Tisch und schubsen es mit einem Stoß in Richtung der x-Achse, so wirkt auf die Testmasse in x-Richtung eine relative Beschleunigung die sich aus der Kraft des Stoßes ergibt. (Das Initialsystem bewegt sich beschleunigt, die Testmasse bleibt träge, wo sie war, und die Feder dehnt sich durch den Stoß …)

ACHTUNG

Die Beschleunigungssensoren in den Android-Geräten messen immer unter Einfluss der Erdbeschleunigung. Die Beschleunigung, die durch Kraftübertragung auf das Gerät wirkt, muss durch eine entsprechende Berechnung, die die Gravitationskraft herausrechnet, ermittelt werden.

Android-Geräte **sollten** einen Beschleunigungssensor haben.

15 In der Kartografie versteht man unter Azimut den im Uhrzeigersinn gemessenen Winkel zwischen geografisch-Nord (Nordpol) und einer beliebigen Richtung (z. B. Marschrichtung, Magnetkompass-Peilung etc.) auf der Erdoberfläche. (Quelle: http://de.wikipedia.org/wiki/Azimuth)

Magnetometer (aka Teslameter oder Gauß-Meter)

Ein Magnetometer ist ein Messgerät, das die magnetische Flussdichte in Tesla (T) misst. Die Sensoren in Android-Geräten liefern Werte mit der Einheit μT (Microtesla). Auch dieser Sensor misst entlang der drei Achsen des Gerätekoordinatensystems, und zwar im Fall unseres Geräts die magnetische Flussdichte des Erdmagnetfeldes. Nun ja, idealerweise des Erdmagnetfeldes, um die Ausrichtung innerhalb dieses Feldes zu bestimmen und damit auch die Ausrichtung gegenüber der Pole, aber das Magnetometer kann nicht unterscheiden, welches Magnetfeld es gerade misst. Befindet man sich in der Nähe großer metallischer Massen oder stromführender Komponenten, ändert sich das Magnetfeld möglicherweise ziemlich stark, und das Magnetometer liefert keine Werte mehr die in wirklicher Relation zum Erdmagnetfeld stehen. Und plötzlich meint unser Gerät, Norden wäre ganz woanders.

Es ist daher ratsam einer Kompassanwendung nur so weit zu trauen, wie man den Magnetometerwerten trauen kann. Es ist durchaus möglich die Verlässlichkeit der Magnetometerwerte abzuschätzen. In meinen Längen- und Breitengraden haben wir eine magnetische Flussdichte von ca. 30 μT. Sollte unser Sensor also stark davon abweichende Werte auswerfen sollte man mal nachschauen, ob nicht irgendwo ein Haufen Metall oder ein altes Telefon oder eine dicke Trafospule in der Nähe ist.

Das Problem hat man aber bei mechanischen Kompassen durchaus auch.

Aber: Idealerweise wissen wir durch die Magnetometerwerte, wie das Gerät im Magnetfeld ausgerichtet ist, und daraus kann man, wenn man nun noch die Orientierung des Geräts berücksichtigt, siehe oben, die Rotation um die z-Achse, die x- und die y-Achse ermitteln.

Glücklicherweise liefert uns Android dafür Hilfsfunktionen, aber auch bei denen muss man sich über die Funktionsweise und vor allem über die Koordinatensysteme im Klaren sein.

INFO

Das Zusammenspiel von Beschleunigungsmesser und Magnetometer für die Ermittlung der Ausrichtung des Geräts in Drehungen in Grad um die Achsen ist ein Beispiel dafür, dass es im Android-System auch sogenannte abgeleitete Sensoren geben kann. Ein Sensor im Framework muss nicht unbedingt genau einem Hardwarebestandteil entsprechen, sondern kann seine Werte von mehreren tatsächlichen Sensoren beziehen. Der mittlerweile veraltete Sensortyp Sensor.TYPE_ORIENTATION ist dafür ein Beispiel.

Android-Geräte **sollten** ein Magnetometer haben.

GPS-Empfänger

GPS-Empfänger dienen der Bestimmung der Position des Geräts auf der Erdoberfläche.

Die Koordinaten werden aus dem Empfang des GPS-Signals mehrerer GPS-Satelliten errechnet. Die GPS-Satelliten senden ständig ihre Position und die genaue Uhrzeit. Der GPS-Empfänger errechnet aus den Signallaufzeiten seine eigene Position als Koordinate im dreidimensionalen Raum. Für die Bestimmung der Position sind mindestens drei Satelliten

erforderlich, zu denen die Entfernung gemessen wird. Die Position des Empfängers liegt auf dem Schnittpunkt der Kugeln um die Satelliten mit dem Radius der jeweiligen Entfernung.

Um die Signallaufzeit allerdings genau zu ermitteln, müssten die Uhren der Satelliten und des Empfängers absolut synchron laufen. Das ist in der Regel aber nicht der Fall, so dass ein weiterer Satellit mit in die Berechnung einbezogen werden muss, um aus den Abweichungen die unbekannte Zeit herauszurechnen.

Wer sich für den genauen Ablauf interessiert, dem sei der Artikel `http://de.wikipedia.org/wiki/GPS-Technik` ans Herz gelegt.

Die Satelliten sind so in der Erdumlaufbahn angeordnet, dass ein Empfänger mindestens vier Satelliten »sehen« kann. Dazu müssen mindestens 24 Satteliten eingesetzt werden, um Ausfällen vorzubeugen, befinden sich zurzeit 31 Satelliten im Orbit.

In der Praxis werden allerdings nicht nur vier Satteliten sondern alle empfangenen Satelliten ausgewertet.

Die Nachrichten der Satelliten werden mit 50 Bit/s gesendet. Eine Navigationsnachricht ist 1500 Bit lang und benötigt somit 30 Sekunden zur Übertragung. In der Nachricht werden die wichtigen Informationen zum Satelliten übertragen, unter anderem die GPS-Zeit zum Sendezeitpunkt und die Bahndaten und mithin die Position des Satelliten.

Parallel zu den Satelliteninformationen überträgt jeder Satellit den sogenannten *Almanach*, der alle Satelliten im Orbit beschreibt. Die Übertragung des Almanachs dauert 12,5 Minuten. Mittels des Almanachs können GPS-Empfänger die Suche nach den Satellitensignalen beschleunigen, da im Almanach Informationen enthalten sind, mit denen der Empfänger bereits Grundannahmen über die Satelliten anstellen kann.

Wenn ein GPS-Empfänger die Satelliten »sucht«, bedeutet das also nicht, dass er nachschauen muss, wo die sich gerade am Himmel befinden, sondern er muss die Signale suchen, die die Satelliten aussenden, und komplett empfangen. Erst zu diesem Zeitpunkt liegt auch die erste verlässliche Messung vor. Diese Zeit wird als *Time to First Fix* (TTFF), jede fertige Positionsmessung wird als *Fix* bezeichnet.

Je nachdem, wie lange der Empfänger abgeschaltet war oder keinen Satellitenempfang hatte und ggf. ohne Satellitenempfang bewegt wurde, kann diese Suche unterschiedlich lang dauern. Um die Zeit zum ersten Fix möglichst zu reduzieren, arbeiten manche Geräte mit *Assisted GPS* (*AGPS*). Diese Geräte können mit einem Almanach, den es z.B. im Internet zum Download gibt, vorinitialisiert werden und sparen sich damit die Zeit, den Almanach von den Satelliten zu empfangen.

Neben der eigentlichen Positionsberechnung kann bei einem Empfänger, der sich in Bewegung befindet, auch die Geschwindigkeit und Bewegungsrichtung ermittelt werden. Diese Berechnung basiert auf dem Dopplereffekt.

Zu weiteren Details siehe `http://de.wikipedia.org/wiki/Global_Positioning_System`.

Aus den dreidimensionalen Koordinaten X, Y und Z berechnet der Empfänger die terrestrischen Koordinaten als Breite und Länge in Grad im *WGS84*[16] (*World Geodetic System1984*). In diesem System läuft der Nullmeridian 100 Meter östlich an der Sternwarte von Greenwich vorbei, die Abweichung ist am Äquator geringer. Das WGS84 ist das weltweit einheitliche Bezugssystem. Die Koordinaten werden auch hier als *Latitude* (geografische Breite) und *Longitude* (geografische Länge) angegeben.

Werden Koordinaten als Pärchen angegeben, dann wird immer die Breite zuerst genannt:

50.484781,8.261649 meint 50.484781° nördliche Breite und 8.261649 östliche Länge.

Die Angaben N/S bei der geografischen Breite zeigen an ob es sich um den Breitengrad der nördlichen oder der südlichen Hemisphäre handelt, die Angabe O/W bzw. E/W (E = East) bei der geografischen Länge zeigt an, ob es sich um die Länge östlich des Meridians oder westlich des Meridians handelt. Obiges Beispiel könnte also auch als 50.484781°N, 8.261649°E notiert werden.

Werden die Angaben N/S und E/W weggelassen, so gilt:

1. Positive Breitengrade sind nördliche Breitengrade, negative Breitengrade sind südliche Breitengrade.
2. Positive Längengrade sind östliche Längengrade, negative Längengrade sind westliche Längengrade.

Die Darstellung der Koordinaten erfolgt in alternativ in:

1. Grad: 50.484781°N, 8.261649°E
2. Minuten: 50°29,08686'N, 8°15,69894'E
3. Sekunden: 50°29'5,2116''N, 8°15'41,9364''

1 Bogenminute entspricht 1/60°, 1 Bogensekunde entspricht 1/3600°.

Das Android-Framework liefert uns einige Methoden, die das Rechnen mit Positionsdaten vereinfachen, um z.B. den Abstand zwischen Koordinaten zu ermitteln. Das könnte für Anwendungen nützlich sein, die Wegrouten aufzeichnen und dann die zurückgelegte Wegstrecke berechnen wollen. Außerdem bieten einige Implementierungen die Möglichkeit, erhaltene Positionsdaten in Adressen oder Adressen in Koordinaten zu wandeln. Das ist z.B. für *Augmented-Reality*-Anwendungen oder zur Anzeige von *Points of Interest* in der Nähe nützlich.

Google liefert außerdem im Android-SDK Klassen für die Nutzung von *Google Maps* auf dem Smartphone mit.

Android-Geräte **sollten** einen GPS-Empfänger haben.

16 http://de.wikipedia.org/wiki/World_Geodetic_System_1984 (abgerufen 12. März 2011, 11:41 MEZ)

INFO

Folgende Stichwörter sind im Zusammenhang mit GPS und der Positionsbestimmung wichtig:

1. **Context.getSystemService(Context.LOCATION_SERVICE);**
2. **LocationManager.getProviders(...);**
3. **LocationManager.setGpsStatusListener(...);**
4. **LocationManager.requestLocationUpdates(...);**
5. **LocationManager.removeUpdates(...);**
6. **LocationManager.removeGpsStatusListener(...);**

Gyroskop (Kreiselinstrument)

Das Kreiselinstrument dient zur Messung der Winkeländerungsgeschwindigkeit. Über die Änderung des Winkels um eine Achse in einem Zeitintervall können wir durch Integration über die Zeit den Winkel bestimmen, um den sich die Orientierung geändert hat.

Mit dem Gyroskop kann man Steuerungen implementieren die sehr genau auf die Änderung der Neigung des Geräts um die Achsen reagieren und sogar die Geschwindigkeit der Änderung mit einfließen lassen.

Bei der Positionsbestimmung kann die Änderung der Winkel in die Berechnung der aktuellen Blickrichtung einbezogen werden, indem nach Kalibrierung der aktuellen Position und Blickrichtung die Winkeländerungen verfolgt und die aktuelle Richtung daraus berechnet wird. Um auch die Positionsänderungen über das Gyroskop nachzuverfolgen, würde man noch die Geschwindigkeit des Geräts benötigen, die man theoretisch aus der linearen Beschleunigung entlang der Achsen ermitteln könnte. Das ist jetzt aber eine gewagte Vermutung die möglicherweise verifiziert werden muss.

Damit wäre man nach einem GPS-Fix eine Zeitlang unabhängig von den Satellitendaten.

Seit Android 2.3 **sollten** Android-Geräte ein Gyroskop haben. Gyroskope sind erst ab 2.3 vorgesehen.

Barometer

Mit dem Barometer wird der Luftdruck gemessen. Hauptsächliche Verwendung findet der Luftdruck in der Berechnung der Höhe über Normalnull.

Seit Android 2.3 **können** Android-Geräte ein Barometer haben.

Thermometer

Googles CDD sagt über Thermometer aus, dass eigentlich keine Thermometer implementiert werden sollen. Und wenn doch, dann nur, um die Temperatur des Prozessors zu messen.

Vergessen wir also das Thermometer.

Android-Geräte **sollten keine** Thermometer haben.

Photometer

Mittels des Photometers kann man die Umgebungshelligkeit in Lux messen. Verwendung kann das Umgebungslicht dabei finden, das User-Interface in den Nachtmodus umzuschalten, z.B. in dem eine Farbpalette gewählt wird, die einen optimalen Kontrast für dunkle Umgebungen liefert.

ACHTUNG

Man kann das aktuelle Umgebungslicht nicht einfach erfragen. Wenn die Anwendung das Umgebungslicht benötigt, muss es sich die Werte liefern lassen und entsprechend darauf reagieren. Da die Werte aber nur bei Änderung geliefert werden, kann es passieren das man nach Start der Anwendung erst einmal eine Zeit lang auf einem nicht bekannten Wert sitzen bleibt. Das Problem kann auch beim Annährungssensor auftreten.

Android-Geräte **können** ein Photometer haben.

Proximity Sensor (Annäherungssensor)

Der Annäherungssensor wird hauptsächlich dafür benutzt die Berührungserkennung und Ähnliches abzuschalten, wenn der Nutzer das Gerät ans Ohr hält. Daher muss, wenn ein Näherungssensor implementiert wird, dieser auch immer in Richtung der z-Achse des Geräts (Blickrichtung des Displays) arbeiten.

Zwar können die Näherungssensoren tatsächliche Abstandswerte in cm liefern, es reicht laut Spezifikation jedoch aus, wenn der Sensor binär arbeitet: 1 für weit, weit weg, 0 für ziemlich nah dran.

Eine weitere Anwendung für den Näherungssensor könnte sein, dass man das Gerät in den Stromsparmodus versetzt oder den Klingelton abschaltet wenn es mit dem Display nach unten auf den Tisch gelegt wird.

Android-Geräte **können** einen Näherungssensor haben. Wenn sie einen haben, dann **muss** er Objekte nahe am Display erkennen.

1.5.4 Netzwerk/Kommunikation

Android-Geräte müssen mindestens eine Form der Netzwerkkommunikation bereitstellen. Es muss ein Netzwerkstandard implementiert sein, der mindestens 200 Kbit/s oder eine schnellere Übertragungsgeschwindigkeit bietet.

Stellt ein Gerät vorzugsweise eine kabelgebundene Netzwerkverbindung bereit, dann sollte das Gerät zumindest einen drahtlosen Standard wie WiFi/W-LAN zusätzlich unterstützen.

Telefonie

Unter Telefonie sind bei Android die Komponenten subsumiert, die Sprachkommunikation und SMS (Short Message Service) über GSM oder CDMA-Netzwerke bereitstellen.

Mit Android 2.3 stellt Android auch Funktionalität für VoIP (Voice over IP) zur Verfügung. Damit ist Telefonie auch über die Datennetzwerkverbindung möglich. Diese Funktionalität gehört aber nicht zur Telefoniefunktion. Telefonie ist in diesem Kontext unabhängig von einer Datenverbindung.

Wenn ein Gerät Telefonieservices anbietet, muss es das in dieser Form tun und Sprache und SMS voll unterstützen. Bietet ein Gerät zwar GSM oder CDMA-Netzfunktionalität an, aber lediglich als Datenverbindung, dann gilt Telefonie als nicht implementiert. Das Framework bietet dann zwar keine Funktion in den Telefonieklassen, die Telefoniefunktionen müssen aber dennoch als sogenannte *No-Ops* (*No operation* – keine Funktionalität) ausgeführt sein. Damit ist sichergestellt, dass Applikationen, die Telefoniefunktionen nutzen, auch auf Geräten ohne Telefonie laufen, z.B. auf Tablets die keine Telefonie, sondern nur Datenverbindung, unterstützen.

Datenverbindungen operieren über TCP/IP per Point-to-Point-Protokoll (PPP). Dazu muss der Mobilfunkanbieter einen PPP-Zugang bereitstellten. Die Bedingungen und Entgelte für die Nutzung der Datenverbindung hängen vom Mobilfunkvertrag ab. Die Geräte lassen sich so konfigurieren, dass Datenverbindungen über GSM/CDMA nie aufgebaut oder beim Roaming im Ausland automatisch abgeschaltet werden. Dennoch sollte man immer ein bisschen aufpassen, über welchen Transport man gerade seine Datenverbindung aufbaut.

INFO

Folgende Stichwörter sind im Zusammenhang mit der Telefonie wichtig:

1. **Context.getSystemService(Context.TELEPHONY_SERVICE);**
2. **TelephonyManager.isRoaming();**
3. **SmsManager.getDefault();**
4. **Interface PhoneStateListener;**
5. **<uses-permission android:name=»android.permission.RECEIVE_SMS« />**
6. **<uses-permission android:name=»android.permission.SEND_SMS« />**
7. **<action android:name=»android.provider.Telephony.SMS_RECEIVED«/>**
8. **Intent.ACTION_CALL**
9. **Intent.ACTION_DIAL**

Android-Geräte **können** Telefoniehardware und Telefoniedienste anbieten.

IEEE 802.11 (WiFi/Wireless LAN)

Mittels WiFi/Wireless LAN (W-LAN) ist eine drahtlose Netzwerkverbindung über sogenannte Access-Points im Infrastruktur-Modus oder auch Ad-hoc-Verbindungen zwischen zwei Teilnehmern möglich. IEEE 802.11 definiert den Medienzugriff und die physikalische Schicht. Auf Basis von IEEE 802.11 können dann unterschiedlichste Netzwerkprotokolle gefahren werden, wobei auf den Android-Geräten ein TCP/IP-Stack realisiert ist.

Je nach Einstellung kann das Betriebssystem die Datenverbindung bzw. die TCP/IP-Kommunikation selbsttätig über die zurzeit beste verfügbare Verbindung leiten. Damit verhält sich das Netzwerk dem Anwender gegenüber transparent. Das ist sehr bequem, aber dafür sollte man einen Flatrate-Datenvertrag haben. Ansonsten sollte man die Einstellung so wählen, dass ein Aufbau der Netzwerkverbindung über GSM/CDMA nicht automatisch erfolgt.

Seit Android 2.2 können Android-Geräte, die WiFi implementieren, selbst als Access-Point dienen und die Datenverbindung über GSM/CDMA anderen WiFi-Clients (wie z.B. einem Laptop) zur Verfügung stellen. Dieses Verfahren wird *Tethering* genannt. Vor Android 2.2 haben manche Hersteller das Tethering selbst implementiert.

Auch hier gilt es den Mobilfunkvertrag zu beachten. Manche Carrier schließen das Tethering vertraglich aus, und wenn keine Datenflat vereinbart ist, sollte man die übertragene Datenmenge genau im Auge behalten.

INFO

Folgende Stichwörter sind im Zusammenhang mit WiF/W-LAN wichtig:

1. **Context.getSystemService(Context.WIFI_SERVICE);**
2. **WifiManager.ACTION_PICK_WIFI_NETWORK**
3. **WifiManager.WIFI_STATE_CHANGED_ACTION**
4. **WifiManager.startScan();**
5. **WifiManager.setWifiEnabled(...);**

Android-Geräte **sollten** WiFi/W-LAN implementieren.

Bluetooth

Bluetooth ist eine Schnittstelle zum Aufbau von *Kleinstnetzen* (*Piconetze*). Hauptzweck bei der Entwicklung von Bluetooth ist der Ersatz von Kabelverbindungen zwischen Peripheriegeräten, z.B. für die kabellose Anbindung von Druckern an einen Computer oder den kabellosen Datenaustausch zwischen Mobiltelefon und Computer. Bekannte Anwendungen sind auch das Headset am Mobiltelefon oder das Streaming von Musik, Bildern oder Videos auf Medienendgeräte.

Für die verschiedenen Anwendungen sind verschiedene Profile definiert. Das Profil beinhaltet die Vereinbarung über das eigentliche Protokoll zwischen den Partnern. Headset-Verbindungen werden z.B. über das A2DP (Advanced Audio Distribution Profile) realisiert.

Android-Geräte **sollten** Bluetooth implementieren.

Near-Field-Kommunikation

Seit Android 2.3 können Geräte auch *Near-Field-Communication* (*NFC*) unterstützen.

Die NFC ist ein Übertragungsstandard zum kontaktlosen Austausch von Daten über kurze Strecken. Kurze Strecken meint wirklich kurz, die Reichweite beträgt maximal 10 Zentimeter mit einer Übertragungsgeschwindigkeit von 424 Kbit/s.

Die kurze Reichweite ist so gewollt. Damit kann eine Kontaktaufnahme zwischen den Partnern als eine gewollte Kontaktaufnahme gewertet werden, und die Gefahr einer unbeabsichtigten Datenübertragung im Vorbeigehen wird minimiert.

Allgemein sollen hier über die kurze Distanz relativ persönliche Daten wie Kontaktdaten, Bilder und andere Informationen zwischen zwei Partnern sehr sicher und nachvollziehbar ausgetauscht werden können.

Das ist eine Voraussetzung, um z.B. Bezahlvorgänge über NFC abzuwickeln (Smartphone als Geldbörse) oder NFC-Geräte als Zugangsschlüssel zu benutzen (Smartphone als Autoschlüssel).

Eine weitere Anwendung liegt im Ersatz von Barcodes durch NFC-Tags. RFID-Tags können z.B. mittels NFC gelesen (und auch beschrieben) werden. Damit ist es möglich Dinge der realen Welt mit Tags zu markieren und diese Tags per NFC-fähigem Gerät zu lesen. Entweder liefert das Tag bereits alle Informationen über das Ding an sich, oder aber über eine eindeutige Kennung können weitere Informationen aus dem Netz geladen werden.

Anwendungsmöglichkeiten wäre z.B. das Markieren (Taggen) von Maschinen und Werkzeugen mit entsprechenden Wartungsinformationen. Oder das Markieren von Ausstellungsstücken, Plakaten, Bildern, Dingen, um nähere Informationen auf das Smartphone abrufen zu können (*Smart Poster*).

Ein sehr lesenswertes Dokument zum Thema NFC und RFID ist unter `http://wwwspies.informatik.tu-muenchen.de/MVS/sem0506/RFID_NFC_folien_ghoefert.pdf` abrufbar[17].

Protokolle und Spezifikationen für NFC-Anwendungen werden vom NFC-Forum (`www.nfc-forum.org`) verwaltet, zu dem viele namhafte Firmen aus dem Hardware-, Netzwerk- und Telekommunikationssektor gehören. Um an die Spezifikationen zu gelangen, muss man sich beim NFC-Forum registrieren und den Lizenzbedingungen zur Nutzung der Spezifi-

17 »RFID und NFC, Technologie, Vergleich und Anwendung«, Gregor Höfert, TU München – Lehrstuhl für Systemarchitektur und Betriebssysteme, Folien zum Seminar »Current Trends in Wireless Networks«, 6.12.2005

kationen zustimmen. Das NFC-Forum erteilt damit die Erlaubnis, die Spezifikation ohne Gebühren innerhalb der eigenen Organisation zu nutzen (Stand März 2011).

Android-Geräte ab Android 2.3 **sollten** NFC implementieren.

1.5.5 Kamera

Rear-Facing Kamera (rückwärtige Kamera)

Die rückwärtige Kamera soll als traditionelle Kamera zum Fotografieren und ggf. zur Aufzeichnung von Videos dienen. Rückwärtig bedeutet, dass die Kamera gegenüber des Displays angeordnet ist, das Display mithin als Sucher dient.

Diese Kamera muss mindestens eine Auflösung von 2 Megapixeln aufweisen.

Die Optik sollte Autofokus bereitstellen, kann aber auch als Festfokus-Optik oder Optik mit erweiterter Schärfentiefe (Extended Depth of Field, EDOF) ausgeführt sein. Der Autofokus kann dabei hardwaretechnisch oder softwaretechnisch realisiert sein, für die Applikationen macht das keinen Unterschied. Das Gerät kann mit einem Blitz ausgestattet sein. Der Blitz wird in einer eigenen Anwendung über die Kameraparameter gesteuert, hiermit ist es z.B. auch möglich, diese witzigen Taschenlampenapplikationen zu realisieren in dem man den Blitz auf »Fackelmodus« schaltet (Camera.Parameters.FLASH_MODE_TORCH).

Android-Geräte **sollten** eine rückwärtige Kamera bereitstellen.

Front-Facing Kamera (frontseitige Kamera)

Die frontseitige Kamera schaut in Richtung des Displays, also dem Anwender ins Gesicht. Die Frontkamera ist hauptsächlich für Videotelefonieanwendungen gedacht, es ist aber auch denkbar, Anwendungen zu schreiben die den Schminkspiegel ersetzen oder in denen man seinem Videopartner einen Schnurrbart malen kann.

Die Mindestanforderungen an die Fronkamera ist VGA-Auflösung mit 640x480 Pixeln, es dürfen aber gerne mehr sein. Die Frontkamera darf niemals die Standardkamera sein, die in der API als Standardkamera zurückgegeben wird. Das ist insofern wichtig, als dass alte Kameraapplikationen, die von der Frontkamera nichts wissen, so ohne Änderung auf neuen Geräten laufen.

Autofokus und Blitz können ebenfalls vorhanden sein und werden genauso angesteuert wie bei der »normalen« Kamera.

Die Voransicht (Preview) wird horizontal gespiegelt, ebenso wie die Bilddaten, die zu den Callbacks geliefert werden. Das muss man beachten, wenn man die Daten selber abfängt und die Voransicht z.B. irgendwohin überträgt. Die endgültige Aufnahme, also entweder das Foto oder der Videodatenstrom, werden allerdings **nicht** gespiegelt, sondern *korrekt* aufgenommen.

Android-Geräte **können** eine frontseitige Kamera bereitstellen.

Allgemeine Eigenschaften der Kameras

Ohne dass wir das Bildformat programmtechnisch vorgeben, liefert die Kamera-API die Bilddaten im YCbCr-Farbmodell in NV21-codiertem Format. Y steht für die Grundhelligkeit, Cb für die Blau-Gelb-Farbigkeit und Cr für die Rot-Grün-Farbigkeit. Die Achsen Cb und Cr spannen also ein Farbspektrum auf, und die Y-Achse bestimmt die Helligkeit der Farbe. Ohne auf die Details eingehen zu wollen, nähert sich dieses Farbmodell dem menschlichen Sehen an, bei dem sich häufig die Farbkomponenten kaum unterscheiden, die Helligkeit (Y) jedoch stärker variiert. Das macht sich z.B. die JPEG-Kompression zunutze, in dem die Abtastrate der Farbigkeit gegenüber der Helligkeit reduziert wird[18].

Dieses Farbmodell wird als Standard in vielen unterschiedlichen Bereichen wie dem Digitalfernsehen, JPEG-Bildern und MPEG-Videos benutzt.

Die Spezifikation schreibt vor, dass die Kamera-API sich immer wohldefiniert verhalten muss, auch wenn die Hardware bestimmte Funktionen nicht bietet. So kann man darauf vertrauen dass die Anwendung z.B. auch dann läuft, wenn kein Autofokus verfügbar ist.

Die Kamera ist immer so orientiert, dass die »lange« Seite des Bildes an der »langen« Seite des Displays ausgerichtet ist, egal wie die »natürliche« Lage des Geräts ist. Das heißt, dass wir für ein Gerät, das natürlicherweise hochkant benutzt wird und bei dem wir unsere Kameraapplikation fest auf Hochkantformat programmieren, die Kamera-Orientierung um 90 Grad nach rechts gekippt werden muss. Bei einem Gerät, das im Querformat genutzt wird und bei dem die Bildschirmausrichtung auch fest auf Querformat gesetzt wurde, muss die Kameraansicht nicht gekippt werden.

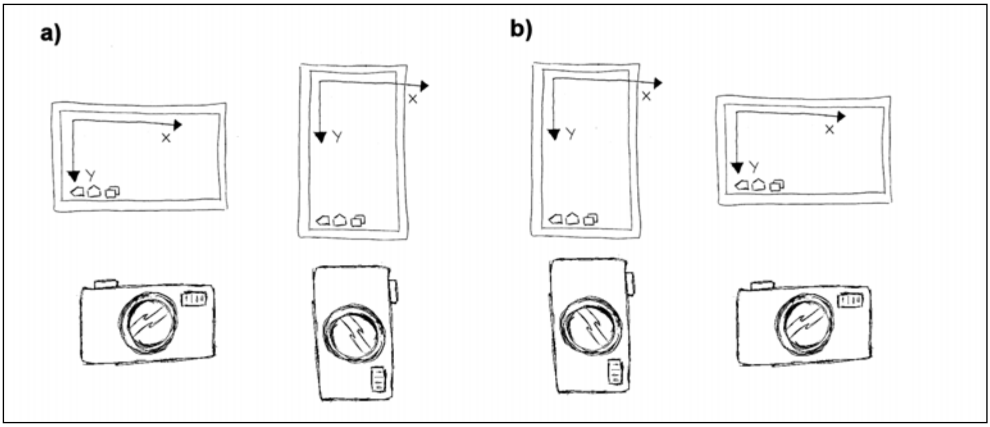


Abbildung 1.8: **Natürliche Ausrichtung der Kamera**

Die Abbildung zeigt in Fall **a)** das Gerät im Querformat als natürliche Ausrichtung. Wenn das Gerät in dieser Ausrichtung betrieben wird, ist das Kamerabild korrekt. Wird das Gerät allerdings gekippt und die Ausrichtung der Kamera nicht angepasst, kippt das Kamerabild

18 Siehe auch http://de.wikipedia.org/wiki/YCvCr-Farbmodell (abgerufen: 13. Januar 2011, 13:28 MEZ)

ebenfalls. Im Fall **b)** handelt es sich um ein Gerät mit Hochformat als natürliche Ausrichtung. Hier ist das Kamerabild ohne Anpassung der Lage bereits im »natürlichen« Betrieb gekippt.

INFO

Ebenso wie beim Auslesen der Sensoren müssen wir beim Ansteuern der Kamera die natürliche Lage und die Lage des Geräts gegenüber der natürlichen Lage berücksichtigen. Die eingebaute Kameraapplikation tut das natürlich von Haus aus, wenn wir unsere eigene Kameraanwendung schreiben, müssen wir darauf achten.

Folgende Stichwörter sind im Zusammenhang mit Kameras wichtig:

1. **SurfaceView**
2. **surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);**
3. **Camera.open(cameraId);**
4. **Camera.Parameters;**
5. **camera.getParameters(...);**
6. **camera.setParameters(...);**
7. **camera.setDisplayOrientation(...);**
8. **camera.setPreviewDisplay(surfaceHolder);**
9. **camera.setPreviewCallback(...);**
10. **camera.startPreview();**
11. **camera.stopPreview();**
12. **camera.release();**
13. **camera.autoFocus(...);**
14. **camera.takePicture(...);**

1.5.6 Speicher

Android-Geräte müssen mindestens 128 MB Speicher für den Betriebssystemkernel und den Ausführungsbereich für Anwendungsprozesse bereitstellen. Die Geräte müssen auch mindestens 150 MB, sollten aber mindestens 1 GB nicht flüchtigen Speicher für Benutzerdaten bereitstellen.

Dieser Speicher für Benutzerdaten ist der Speicher, in dem unsere Applikation private Daten ablegen kann, entweder als `SharedPreferences` oder durch direkten Zugriff auf diesen internen Speicher via `openFileOutput(…);`

INFO

Der interne, applikationsbezogene Speicher wird beim Deinstallieren der Applikation ebenfalls gelöscht. Das ist zu berücksichtigen, falls dort »wichtige« Daten gespeichert werden.

Neben dem internen Speicher müssen Android-Geräte mindestens 1 GB an gemeinsam nutzbarem Speicher (Application Shared Storage, External Storage) bereitstellen. Auf den gemeinsam nutzbaren Speicher können alle Applikationen gleichermaßen zugreifen, die Daten dort sind also nicht privat. Außerdem muss dieser Speicher beim Anschluss des Geräts an einen Computer via USB per Media-Transfer-Protokoll auf dem Computer zum Zugriff freigegeben werden, sodass die Inhalte des Dateisystems auch über einen Computer verwaltet werden können.

ACHTUNG

Wenn das Dateisystem durch Anschluss an einen Computer dort eingebunden wird, dann haben die Android-Applikationen während dieser Zeit möglicherweise keinen Zugriff mehr darauf. Vor Verwendung des externen Speichers sollte unsere Anwendung also den Status des Dateisystems prüfen und ggf. bestimmte Funktionen abschalten, die zurzeit nicht auf das Dateisystem zugreifen können.

Auf dem Application Shared Storage können wir zum einen externe Daten speichern, die bei der Deinstallation unserer Anwendung **ebenfalls gelöscht** werden. Das Verzeichnis für unsere Applikation erhalten wir über `getExternalFilesDir()`.

Zum anderen können wir auch gemeinsam genutzte Daten speichern die bei der Deinstallation **nicht gelöscht** werden. Dazu benutzen wir das entsprechende Verzeichnis das wir mittels `getExternalStoragePublicDirectory(…)` erhalten.

Android gibt einige öffentliche Verzeichnisse vor, die für unterschiedliche Inhalte benutzt werden können und vom Mediascanner durchsucht werden, um z.B. Musik und Bilder den eingebauten Applikationen bekannt zu machen.

Android stellt mittels `getExternalCacheDir()` auch ein spezielles Cache-Verzeichnis zur Verfügung in das wir Daten ablegen können, die zu Auslagerungszwecken oder zum Zwischenspeichern gedacht sind. Auch dieses Verzeichnis wird bei der Deinstallation der Applikation gelöscht. Die Lebensdauer der Dateien, während unsere Applikation installiert ist, müssen wir aber selbst verwalten, die Dateien werden z.B. nicht automatisch gelöscht, wenn die Applikation geschlossen wird.

INFO

Folgende Stichwörter sind im Zusammenhang mit Speicher wichtig:

1. **SharedPreferences**
2. **getSharedPreferences(…);**
3. **openFileOutput(…);**
4. **getExternalStorageState();**
5. **getExternalFilesDir();**

INFO

6. **getExternalStoragePublicDirectory();**
7. **getExternalCacheDir();**
8. **File, FileInputStream, FileOutputStream**

1.5.7 SQL-Datenbank

Das Android-Betriebssystem bietet mit SQLite eine eingebaute SQL-Datenbank. Damit ist es möglich, strukturierte Daten abzulegen (z.B. Adressen, Notizen, Karteikarten, Termine) und per SQL die Daten zu verwalten, also zu suchen, neue Einträge zu erstellen etc.

Die SQL-Datenbanken sind immer privat zur Anwendung. Der Zugriff auf die Datenbank kann anderen Anwendungen mittels Content-Providern gewährt werden.

TIPP

Es ist eine gute Idee, strukturierte Daten, und bestehen sie auch nur aus einem Datum, einem Titel und einem Text, in der SQL-Datenbank abzulegen. Zum einen braucht man sich keine Gedanken um ein adäquates Dateiformat zu machen, zum anderen bietet SQL einen wirklich bequemen und schnellen Zugriff auf die strukturierten Daten, den man mit einer eigenen Datenstruktur erst einmal realisieren müsste. Selbst die Verwaltung großer binärer Objekte wie Bilder etc. lässt sich in SQLite sehr effizient bewerkstelligen. In die Überlegung der Speicherung muss aber auch mit einbezogen werden, ob die Daten z.B. via USB direkt verwaltet werden können müssen/sollen, und die Synchronisierung der SQLite Datenbank z.B. zu Datenaustausch- und/oder Backup-Zwecken muss berücksichtigt werden.

1.5.8 Synchronisieren und Backup

Eine wichtige Funktion ist selbstverständlich Backup und das Synchronisieren von Daten.

Beide Methoden basieren auf dem Gedanken von »Cloud«-Services, d.h., die Partnersysteme für Backup und Synchronisierung liegen irgendwo im Internet. Der Backup-Service ab Android 2.2 ist sehr eng mit Google verdrahtet. Google stellt die Backup-Server zur Verfügung und man muss seine Applikation beim Backup-Service registrieren. Wir erhalten dann einen Schlüssel, mit dem wir unseren Backup-Agenten signieren. So ausgestattet können wir in unserer Applikation beim Betriebssystem ein Backup oder ein Restore anfordern, das dann über den Backup-Manager durchgeführt wird. Innerhalb des Backup-Agenten bestimmen wir, was gesichert bzw. zurückgesichert werden soll, und übergeben die Daten dann transparent an den Backup-Service. Der Service wiederum übernimmt die Ablage bzw. das Holen der Daten von den Backup-Servern.

ACHTUNG

Die API des Backup-Services ist sehr schick gemacht, ich finde allerdings die enge Verzahnung mit Google an der Stelle etwas problematisch. Eigentlich möchte ich das Backup auf eigenen Servern ablegen und selbst die Kontrolle darüber behalten. Außerdem ist es momentan so, dass ggf. nicht jedes Gerät den Backup-Transport anbietet oder sogar anders implementiert.

Für die Synchronisierung von Daten unserer Applikationen gibt es da schon bessere Möglichkeiten. Grundsätzlich können Synchronisierungen mittels *Sync-Adaptern* realisiert werden, die wir für unsere Anwendung und unseren eigenen Dienst bereitstellen. Innerhalb der *Sync-Adapter* sind wir vollkommen frei zu wählen, welche Daten wir gegen einen anderen Datenbestand abgleichen. Über einen eigenen *Authenticator* können wir den Anmeldemechanismus an unserem Serverdienst im Betriebssystem verankern. Unsere Synchronisierung benutzt diese Informationen, um sich an unserem eigenen Dienst anzumelden. Wenn wir also eine eigene Adressdatenbank haben, die wir mit den Kontakten auf dem Smartphone synchronisieren wollen, dann können wir das über diesen Mechanismus erledigen.

TIPP

Da wir hier die Chance haben, einen eigenen Dienst anzusprechen, und nicht an einen Service von Google oder anderen gebunden sind, erscheint mir dieser Weg des Datenaustauschs auch für eine Backup-Möglichkeit momentan etwas besser. Selbstverständlich müssen wir uns hier um den Transport und das Speichern selber kümmern, das ist also aufwendiger.

Die Sync-Adapter sind aber ein guter Ansatzpunkt für unseren persönlichen Datenabgleich.

1.5.9 USB

Android-Geräte müssen einen USB-Anschluss aufweisen und die Debug Bridge over USB implementieren. Das ist für uns Entwickler eine gute Sache, denn dadurch können wir uns darauf verlassen, jedes Android-Gerät zum Entwickeln und Testen in unsere Entwicklungsumgebung einbinden zu können.

Weiterhin muss die USB-Massenspeicherspezifikation implementiert werden, damit der Inhalt des externen Speichers über den Host, an den das Gerät angedockt wird, verwaltet werden kann.

INFO

Leider ist es, zumindest in der Standardlaufzeitumgebung, nicht so einfach möglich eine Kommunikationsverbindung zum Host über die USB-Schnittstelle herzustellen, um z.B. eigene Synchronisierungssoftware bzw. einen eigenen Synchronisierungsserver bereitzustellen. Dazu müsste man wohl entweder die Android-Debug-Bridge missbrauchen oder aber über das Native Development Kit (NDK) einen entsprechenden Gerätetreiber implementieren. Möglicherweise kann man einen Workaround schaffen, in dem man das USB-Tethering verwendet und darüber auf den Host zugreift. In Android 3.1 werden bezüglich der USB-Konnektivität weitere Verbesserungen implementiert, um Geräte per USB an das Gerät anzuschliessen und die USB Verbindung zu benutzen.

1.6 Der innere Kern

Der vorherige Abschnitt beschäftigt sich im Großen und Ganzen mit der üblichen bzw. möglichen Hardwareausstattung von Android-Geräten auf dem API-Level 10 bzw. 11. Neben den Anforderungen an die Hardware liefert das CDD noch weitere Spezifikationen bezüglich des

Betriebssystems und der Android-Laufzeitumgebung, aus der wir wertvolle Informationen über immer verfügbare Standardimplementierungen finden können.

Dabei ist es besonders interessant, dass Android in der Laufzeitumgebung für viele Dinge bereits Standardanwendungen bzw. Standardmodule zur Verfügung stellt:

1. Schreibtischuhr
2. Browser
3. Kalender
4. Taschenrechner
5. Kontaktverwaltung (Adressen, Telefonnummern etc.)
6. E-Mail
7. Bildgalerie und Kamera
8. Übergreifende Suche
9. Startbildschirm(e)
10. Musik
11. Einstellungen

Das heißt, wir können uns als Anwender und Entwickler darauf verlassen, dass unser Android-Gerät diese Anwendungen implementiert und zur Verfügung stellt und dass sich alle diese Anwendungen mindestens so verhalten wie durch die Spezifikation vorgegeben.

Da auch die Emulatoren der Entwicklungstools diese Spezifikation erfüllen müssen, wissen wir, dass in jedem Android-Grundsystem diese Anwendungen vorhanden sind.

Allerdings erlaubt die Spezifikation auch, dass ein Hersteller eines Android-Geräts Teile der Standardanwendungen oder auch alle Anwendungen durch eigene Implementierungen austauschen kann. Das machen einige Hersteller, um eigene Startbildschirme, eigene E-Mail-Clients bis hin zu einer ganz eigenen Oberfläche zu realisieren, die möglicherweise über den Standard hinausgehen. Aber: Wir können uns wie beschrieben immer darauf verlassen, dass selbst die ausgetauschten Anwendungen mindestens die Funktionalität der Standardanwendung bereitstellen. Somit ist gewährleistet, dass unsere Anwendungen auf jedem System immer die gleiche Standardfunktionalität vorfinden und auch entsprechend lauffähig sind.

INFO

Nicht nur die Hersteller der Geräte, auch wir können alternative Anwendungen für die Standardanwendung bereitstellen. Meistens können wir die Standardanwendung nachträglich nicht austauschen, aber Android bietet ein Konzept, um aus vorhandenen Alternativen auszuwählen und damit dem Anwender die Wahl zu lassen, unsere Anwendung zu nutzen oder die andere(n) Anwendung(en).

1.6.1 Modularisierung und Kopplung

Android benutzt sogenannte *Intents* um eine *lose Kopplung* von Applikationen zu erreichen. Unter *loser Kopplung* versteht man eine Technik, bei der die unterschiedlichen Anwendungen keine direkte Bindung an andere Anwendungen haben. Bei einer direkten Bindung würde eine Anwendung z.B. direkt auf Funktionen einer Bibliothek oder einer anderen Anwendung zugreifen und müsste diese Funktionen sowie ihre Signatur, also das Aussehen, ganz genau kennen. So etwas birgt immer die Gefahr von Inkompatibilitäten. Wenn sich z.B. die Signatur einer Funktion ändert, müsste auch der Aufrufer entsprechend geändert werden. Außerdem kann dann wirklich nur eine Bibliothek diese Funktion zur Verfügung stellen, eine Erweiterung des Systems um alternative Implementierungen ist nicht so einfach möglich. Um diese Probleme zu minimieren und ein System sehr einfach erweiterbar zu gestalten, implementiert man im Betriebssystem oder dem Laufzeitsystem einen Kommunikationskanal, über den die Anwendungen Nachrichten verschicken und Nachrichten empfangen können. Das Format solcher Nachrichten ist durch das Laufzeitsystem standardisiert, und die »einzige« Funktion, die die Anwendungen kennen müssen ist die Funktion zum Verschicken und Empfangen von Nachrichten.

In Android ist dieses System mittels der *Intents* realisiert.

Alle Standardanwendungen im Android-System besitzen einen vordefinierten Satz an Intents, sogenannte *Intent-Patterns*. Damit ist vorgegeben, auf welche Nachrichten mit welchen Parametern die Anwendungen reagieren und welche Nachrichten sie selbst verschicken können (auf die wiederum andere Anwendungen reagieren können).

Die Modularisierung ist ein zentraler Aspekt des Android-Systems und darf von Geräteherstellern **nicht** eingeschränkt werden. Das bedeutet, dass wir immer und auf jedem Android-Gerät die Standardimplementierungen vorfinden – egal wie sie ausgeführt sind – und auch unsere eigene Implementierung bereitstellen können, falls uns diese besser gefällt. Android ermöglicht dann dem Nutzer, mittels des *Chooser*-Dialogs aus den Alternativen auszuwählen. Auch diesen *Chooser* darf ein Hersteller **niemals** unterbinden.

Für uns bedeutet dies, dass wir Standarddienste oder auch Dienste anderer Applikationen (in der Regel) immer mittels Intents nutzen.

Intents dienen wie beschrieben dazu, eine ACTION einer Anwendung auszulösen. Das passiert über unterschiedliche Parameter bzw. eine Kombination daraus:

1. Den Namen der Zielkomponente
2. Den Namen der auszuführenden Aktion (ACTION)
3. Der Datentyp und die Daten, für den/mit denen eine Aktion ausgeführt werden soll

INFO

Nebenbei bemerkt dienen die *Intents* nicht nur dazu, Aktivitäten in anderen Anwendungen auszulösen, sondern auch dazu, Aktivitäten in unserer eigenen Anwendung auszulösen. Das Intent-System ist also wirklich ein sehr, sehr zentraler Mechanismus des Android-Laufzeitsystems.

Bemerkenswert ist der dritte Aspekt. Das Android-System definiert z.B. eine Standardaktion ACTION_EDIT. In Verbindung mit dem Datentyp bzw. einer Datenquelle können wir damit, wenn denn nun eine entsprechende Anwendung installiert ist, aus unserer Anwendung eine Bearbeitungsmöglichkeit für die Daten aufrufen, ohne selbst irgendetwas über diese Applikation wissen zu müssen.

Datentypen werden als MIME-Typen (image/jpeg, text/plain, ...) angegeben, Datenquellen als URI.

Ein schönes Beispiel ist die Nutzung der eingebauten Galerie, um ein JPEG-Bild für die eigene Anwendung auszuwählen. Hier bedient man sich der Aktion ACTION_GET_CONTENT für den MIME-Typ image/jpeg. Über das Intent-System wird die Activity herausgesucht, die auf die Aktion für den MIME-Typ antworten kann, und das wird entweder die eingebaute Galerie oder eine Auswahl an alternativen Activities sein.

Die *Intents* werden auch dafür benutzt sogenannte *Broadcast Events* auszulösen. *Broadcast Events* werden z.B. ausgelöst, wenn sich an der Hardware- oder der Softwarekonfiguration etwas ändert, die Batterie schwach wird, das Gerät an einer Dockingstation angeschlossen wird etc. Wir können mittels *Broadcast Receivern* auf solche Ereignisse in unserer Anwendung reagieren. Genau wie Standardanwendungen definiert das Android-System einen Satz von vorgegebenen *Broadcast Events*, die wir in jeder Implementierung vorfinden werden und die sich auch immer gleich verhalten.

1.6.2 Die Benutzeroberfläche

Android definiert in der Laufzeitumgebung eine Menge an Elementen, um die Benutzeroberfläche zu gestalten.

Zu beachten ist bei der Betrachtung der Benutzeroberfläche, dass es sich hier nicht um ein traditionelles Fenstersystem mit frei verschiebbaren Fenstern wie unter Windows handelt, aufgrund des geringeren Platzes gar nicht handeln kann.

Die Benutzeroberfläche ist als Stapel ausgeführt, bei dem sich die einzelnen Aktivitäten wie Spielkarten über die anderen legen. Das Interaktionsschema bei Smartphones und auch bei Tablets ist weniger das parallele Nutzen von Fenstern als das Vor- und Zurückbewegen bzw. das Durchblättern dieses Stapels. Daher kommt einem *Taskmanager* bzw. einer *Recent Apps*-Liste eine große Bedeutung zu. Bis Android 2.3.3 waren die aktuell laufenden Anwendungen nicht so einfach zu erreichen, meist über spezielle Anwendungen, ab Android 3.0 liefert die *System Bar* einen schnellen Zugriff auf die gerade laufenden Anwendungen, sodass ein Umschalten zwischen den Anwendungen schnell vonstatten geht.

Neben Elementen zur Layoutgestaltung und vordefinierter Elemente wie Textfelder, Eingabefelder, Knöpfe, Menüs etc. muss jedes Android-Gerät bestimmte Mechanismen zur Verfügung stellen, die die elementaren Funktionen der Benutzeroberfläche darstellen, und die jeder Gerätehersteller auch in abgewandelten Oberflächen unterstützen muss.

Die Oberfläche des Android-Geräts besteht in der Regel immer aus der *Status Bar* bzw. bei Tablets ab Android 3 aus der *Action Bar*, die am oberen Bildschirmrand angeordnet ist, sowie ab Android 3 aus der *System Bar*, die am unteren Bildschirmrand angeordnet ist. In wieweit die System Bar auch bei Smartphones zum Einsatz kommen wird, muss die weitere Entwicklung zeigen, ggf. wird auch bei Smartphones zunehmend auf Hardwaretasten verzichtet und möglicherweise mit der System Bar gearbeitet.

Dazwischen befindet sich der für Anwendungen nutzbare Bildschirmbereich.

Android gestaltet das Aussehen aller Elemente konsequent über sogenannte *Style-Ressourcen*, in denen Farben, Schriftgrößen und Schriftstile, Hintergrundbilder etc. für die einzelnen Elemente definiert werden. Diese Stile werden in *Themes* zusammengefasst. Zwischen den *Themes* kann gewechselt und damit das Aussehen der Oberfläche entweder komplett oder speziell für einzelne Anwendungen verändert werden. Mit den Themes ist es z.B. auch möglich, »Vollbildanwendungen« zu erstellen, bei denen die Zusatzleisten ausgeblendet werden und damit der verfügbare Platz größer wird. Das wird z.B. für Spiele gern gemacht.

Widgets

Ein Widget ist im Allgemeinen ein Oberflächenelement, das ein bestimmtes Aussehen und auch eine bestimmte Funktion hat. Ein Button ist z.B. ein Widget, das einen Knopf darstellt, auf den man drücken kann.

Android führt noch das Konzept der sogenannten *AppWidgets* ein. *AppWidgets* sind kleine Bereiche auf dem Startbildschirm, die selbst ein kleines Programm darstellen. Bis Android 2.3.3 sind die AppWidgets hauptsächlich dafür gedacht, irgendetwas Aktuelles wie die Uhrzeit, das Wetter, die Aktienkurse und die Anzahl entgangener Anrufe anzuzeigen. In Android 3.0 wurden die AppWidgets erweitert, um auch komplexere Inhalte darzustellen und mehr Interaktion direkt mit dem AppWidget zu ermöglichen.

Die App-Widgets sind, bei dieser Art der Benutzeroberfläche, die einzige Möglichkeit bestimmte Dinge unterschiedlicher Anwendungen überhaupt parallel ansehen und benutzen zu können, ohne zwischen den Ansichten hin- und her zu schalten.

AppWidgets sind als *RemoteViews* realisiert, da sie praktisch innerhalb einer anderen View (der HostView) eingebettet werden, die Inhalte aber von der RemoteView-Anwendung kommen.

Benachrichtigungssystem/Notifications

Ein weiterer zentraler Bestandteil sind Mechanismen zur Benachrichtigung des Benutzers.

Android bietet für unterschiedliche Zwecke standardisierte Benachrichtigungen an:

1. Toast Notifications
2. Status Bar Notifications
3. Dialog Notifications

Neben diesen rein visuellen Benachrichtigungen können noch akustische und haptische Benachrichtigungen realisiert werden:

1. Vibration
2. Alarmtöne und Signaltöne
3. Lichtsignal (blinkende LED, Helligkeit des Bildschirms verändern)

Bei visuellen Benachrichtigungen ist es immer wichtig zu unterscheiden, warum und was man mitteilen möchte.

Eine *Toast*-Benachrichtigung wird als Nachricht in dem Moment auf dem Bildschirm angezeigt, zu dem sie ausgelöst wird, und verschwindet nach einer gewissen Zeit wieder. Diese Benachrichtigung wird gern für die Bestätigung einer Aktion genutzt, z.B. wenn ich einen Alarm eingestellt oder Daten abgespeichert oder ein Bild aufgenommen habe, um anzuzeigen, dass die Aktion jetzt abgeschlossen ist. Egal wer den Toast auslöst, also unsere Anwendung oder ein Hintergrundservice, die Nachricht überlagert immer für kurze Zeit die aktuelle Anwendung. Ein Toast bietet aber keine Möglichkeit der Interaktion, das heißt er kann nicht angetippt oder vorzeitig geschlossen werden.

Die *Status Bar Notification* dient dazu, Benachrichtigungen anzuzeigen, die *irgendwann* eine Reaktion des Benutzers erfordern und deshalb über längere Zeit erreichbar sein müssen. Die *Status Bar Notification* erlaubt das Platzieren eines Icons und einer Nachricht, die als Ticker ausgeführt wird, in der *Statusbar* sowie einer ausführlichen Nachricht im zugehörigen Statusfenster. Das Statusfenster kann man aus der Statusbar wie ein Rollo hinunterziehen oder heraufziehen, und dort wird eine Liste der aktiven Nachrichten gesammelt. Mit der Nachricht kann auch eine Aktion verknüpft, d.h. auf das Antippen der Nachricht kann reagiert werden. Diese Form der Benachrichtigung wird gerne verwendet, wenn ein Ereignis im Hintergrund passiert, z.B. eine Terminerinnerung fällig wird, ein Download abgeschlossen ist oder ein Anruf verpasst wurde, und der Benutzer zu irgendeinem Zeitpunkt später sich darum kümmern können soll. Zur eigentlichen Nachricht kann man den Benutzer zusätzlich mit Vibration, Sound oder auch Lichtsignalen (blinkende LED) benachrichtigen.

Im Gegensatz dazu werden *Dialog Notifications* dann genommen, wenn die Anwendung im Vordergrund gerade zum jetzigen Zeitpunkt etwas tut und den Benutzer direkt benachrichtigen muss oder die Anwendung eine länger dauernde Aufgabe durchführt und dem Benutzer während dieser Zeit einen Fortschrittsdialog, ggf. mit Abbruchmöglichkeit, zeigen will. Dialoge können im Gegensatz zu *Toast*- und *Staus-Bar*-Nachrichten auch wesentlich komplexer gestaltet werden, z.B. eine Auswahlliste, einen ausführlichen Text und verschiedene Buttons anzeigen.

Suche

Die übergreifende Suche ist ebenfalls ein zentraler Bestandteil des Systems. Die Suche ist als systemweites Benutzerinterface ausgeführt, in das sich die unterschiedlichen Anwendungen einklinken können. So können wir die Suche entweder innerhalb unserer Anwendung mit unseren Daten nutzen, aber auch unsere Daten in die systemweite Suche einspeisen.

INFO

»Früher« organisierte man sich noch hauptsächlich in traditionellen Ordner- und Dateistrukturen. Der systematische Mensch sucht in seinen Aktenschränken nach der richtigen Akte. Außerdem arbeitete man doch noch sehr anwendungszentriert, das heißt man öffnet zuerst die Anwendung und sucht dann nach den richtigen Dateien. Gerade bei Datenbankprogrammen musste man ja sowieso erst die richtige Anwendung starten. Mit zunehmender »Googleisierung« der Gesellschaft weicht diese Systematik aber eher der »chaotischen« Suche, bei der ich nicht mehr definiere, wo ich suche (Ordner und Dateien), sondern was ich suche (Themen, Stichwörter). Wie die Daten organisiert sind und mit welcher Anwendung ich diese Daten bearbeite, tritt somit mehr in den Hintergrund. Daher ist diese systemweite Suche in Android eine konsequente Übertragung dieses Web-Prinzips auf unsere mobilen Datensammlungen, bei der wir nicht mehr unbedingt wissen, ob die Daten auf dem Gerät oder irgendwo in der Wolke liegen.

Für uns bedeutet das wiederum, dass wir auf allen Implementierungen den systemweiten Suchmechanismus vorfinden und tunlichst keine eigene Art und Weise der Suche implementieren sollten.

Die Spezifikation schreibt auch vor, dass die Suche über einen ständig verfügbaren Mechanismus, z.B. ein globales Suchfeld in der Actionbar, mindestens jedoch über eine Hard- oder Softwaretaste zu erreichen ist.

Live Wallpapers

Live Wallpaper sind lebende Hintergründe, die auf den Startbildschirmen als Hintergrund ausgeführt werden. Ein lebender Hintergrund ist einem Hintergrunddienst (Service) nicht unähnlich, da er praktisch im Hintergrund abläuft. Im Unterschied zu einem Service kann der lebende Hintergrund aber auf den Hintergrund der Startbildschirme zeichnen und eine gewisse minimale Form der Interaktion mit dem Benutzer zulassen.

1.6.3 WebKit und HTML 5

Ein ganz wichtiges Merkmal der mobilen Gerätegeneration ist ja die mobile Internetnutzung. Neben der eigentlichen Telefonie (bei Smartphones) ist ja die ständige, mobile Verfügbarkeit des Internets und damit von E-Mail, Social-Networks und Webseiten ein wesentlicher Anwendungsaspekt der Geräte.

Außerdem entwickelt sich die Netzwelt ja immer mehr dazu, nicht nur Inhalte anzubieten, sondern, siehe Social-Networks, höchst interaktive Anwendungen bereitzustellen. Und, darüber hinaus, ist es heute gang und gäbe, Rich Content per HTML und dynamischen Content per HTML und JavaScript zu erstellen und in seine eigenen Anwendungen zu integrieren, ja geradezu möglichst komplette Anwendungen mit dieser Technik zu entwickeln.

Diesem Trend folgt auch der aktuelle Standardisierungsprozess für HTML 5, das einige Erweiterungen erhält, um über eine reine Dokumentbeschreibungssprache hinaus zu wachsen, und die Entwicklung von Anwedungen unterstützt. Einige bemerkenswerte Eigenschaften sind die Möglichkeit lokaler Datenspeicherung, native 2D-Grafiken, Audio-/Video-/Image-Support und Geolocation.

Dafür müssen die mobilen Endgeräte einen entsprechenden Browser mitbringen bzw. im System entsprechende Komponenten zur Verfügung stellen. Android schreibt hier die Implementierung von WebKit vor. Und das ist eine ziemlich tolle Sache, denn WebKit ist eine freie Bibliothek (LGPL und BSD Lizenz), die von, und jetzt kommt´s: Apple, Google, Nokia, Adobe, KDE und anderen gepflegt und weiterentwickelt wird. Originär stammt der Code aus dem KDE-Projekt, und wurde von Apple als Grundlage für Safari weiterentwickelt und in das WebKit-Projekt überführt.

Warum ist das nun so bemerkenswert? Wenn man sich die Entwicklung von WebKit auch bezüglich der JavaScript-Implementierung anschaut, dann kann man mit Browsern bzw. Komponenten, die auf WebKit basieren, heute schon (kleine) Anwendungen bauen. Berücksichtigt man jetzt noch die Verbreitung von WebKit-Browsern, dann haben wir mit HTML5 in Verbindung mit WebKit eine Möglichkeit, plattformübergreifend mobile Anwendungen zu entwickeln, denn WebKit werkelt in iOS, in Android, bei Nokia, Openmoko, Palm Pré und so weiter.

Für uns bedeutet das, dass wir eine vollwertige HTML-/JavaScript-Komponente in unseren Anwendungen nutzen können, um Rich-Content-Oberflächen zu gestalten und möglicherweise Teile unserer Anwendung als »Webanwendung« zu realisieren.

1.6.4 Multimedia

Jedes Android-Gerät muss eine Möglichkeit der Audio-Ausgabe bieten, sei es über interne Lautsprecher, Kopfhörer oder externe Lautsprecher. Die Möglichkeit der Audio-Aufnahme, das heißt ein Mikrofon, muss nicht realisiert sein.

Dennoch müssen alle Geräte die Multimedia-API komplett implementieren.

Für das Abspielen von Inhalten (Video, Audio und Bilder) muss jedes Gerät einen Satz an Decodern implementieren, die die gebräuchlichsten Formate beinhalten. Für uns als Entwickler bedeutet das nun wiederum, dass wir uns darauf verlassen können, diese Decoder auf allen Geräten vorzufinden.

Für das Aufzeichnen von Inhalten (Audio, Video, Bilder) muss jedes Gerät ebenfalls einen Satz an Encodern bereitstellen. Zwar kann es Geräte geben die kein Mikrofon und/oder keine Kamera bereitstellen, dennoch muss zumindest der Encoder für JPEG und PNG Dateien implementiert sein, da Bilder in diesem Format auch softwaretechnisch erzeugt werden können. Man stelle sich z.B. eine Fingermalapplikation vor, mit der man Bilder im PNG- oder JPEG-Format abspeichern möchte.

1.6.5 Sicherheitsaspekte

Es ist offensichtlich, dass es auf einem Gerät, das potenziell am Internet hängt, mit dem man telefonieren und SMS schreiben kann, das über ein Mikrofon und eine Kamera sowie GPS Sensor verfügt, ziemlich unerwünscht ist wenn eine Anwendung einfach so ungefragt auf diese Funktionen zugreifen kann.

Außerdem sollen natürlich Anwendungen auch nicht so ohne Weiteres auf private Daten wie die Kontakte oder E-Mails und gespeicherte SMS zugreifen können.

Daher realisiert Android ein relativ umfangreiches Sicherheitssystem.

Grundlage des Sicherheitssystems sind zum einen die User-ID, die jeder Anwendung bei Installtion zugewiesen wird, und zum anderen die Gewährung von Zugriffsrechten auf bestimmte Systemressourcen durch den Anwender.

Eine Anwendung, die auf einem Android-Gerät installiert werden soll, muss mit einem Zertifikat versehen werden, das mit dem privaten Schlüssel des Entwicklers erstellt wurde. Den Schlüssel zum Signieren der Zertifikate erhält man, wenn man sich beim Android-Market registriert, und das ist auch Voraussetzung, um Applikationen über den Market zu vertreiben.

Das Zertifikat können wir selbst ausstellen, das muss nicht von einer offiziellen Zertifikatsautorität gemacht werden. Das Erstellen des Zertifikats können wir ebenfalls mit den Entwicklertools durchführen.

In unserer Entwicklungsumgebung werden die Anwendungen mit einem Entwicklerschlüssel signiert, der die Installation via USB mit den Entwickletools erlaubt, sodass wir uns für das Entwickeln nicht beim Android-Market registrieren müssen.

Das Zertifikat selbst dient dazu, die Anwendungen einem Author zuzuordnen. Mit dem Zertifikat können bestimmte Sicherheitsmechanismen implementiert werden, damit nur Anwendungen mit dem gleichen Zertifikat (also vom gleichen Autor) die Erlaubnis für eine Aktion erhalten bzw. um die Nutzung einer gemeinsamen User-ID und damit den gemeinsamen Zugriff auf private Daten für Anwendungen desselben Autors zu ermöglichen.

Android definiert für die hier besprochenen Komponenten (Hard- und Software) entsprechende Berechtigungen, z.B. `android.permission.CAMERA` um Zugriff auf die Kamera zu erhalten. Wenn wir in unserer Anwendung auf die Kamera zugreifen wollen, dann müssen

wir das im Manifest der Anwendung bekannt machen, ansonsten verweigert die Laufzeitumgebung den Zugriff auf die Kamera.

Wenn ein Benutzer dann unsere Anwendung aus dem Android-Market installieren will, sieht er, dass unsere Anwendung auf die Kamera zugreifen möchte. Bei der Installation wird er nun gefragt, ob er damit einverstanden ist.

Dadurch hat der Anwender die Möglichkeit, selbst zu entscheiden was die Anwendungen auf seinem Gerät anstellen dürfen.

1.7 Die Oberfläche

1.7.1 Hochformat und Querformat

Wie wir bereits bei der Betrachtung des Bildschirms, der Sensoren und der Kamerahardware erfahren haben, besitzen alle Geräte eine natürliche Ausrichtung. Smartphones wie das Google Nexus S, die HTC-Serie und viele andere werden in der Regel im Hochformat betrieben. Die Tablets wie das Motorola XOOM oder das Acer ICONIA sind in der Regel im Querformat konzipiert.

Das Betriebssystem kann auf die Änderung der Lage des Geräts gegenüber seiner natürlichen Lage reagieren und die Benutzeroberfläche entsprechend anpassen. Wie bereits beschrieben, wird dabei das Koordinatensystem angepasst, so dass die y-Achse wieder nach unten und die x-Achse nach rechts zeigt. Der Koordinatenursprung verbleibt dabei in der linken oberen Ecke.

Wir müssen diesem Umstand dadurch Rechnung tragen, dass wir unsere Layouts entsprechend flexibel gestalten, damit unsere Anwendung in beiden Ausrichtungen korrekt dargestellt wird. Weiterhin müssen wir ggf. die Auswertung der Sensoren und die Ausrichtung der Kamera entsprechend anpassen.

Das Layout- und Resourcensystem von Android erleichtert das Gestalten solch flexibler Oberflächen. Wir können Layouts so anlegen, dass sie sich dynamisch an die Bildschirmgröße anpassen, und wir können für unterschiedliche Bildschirme, unterschiedliche Auflösungen und unterschiedliche Ausrichtungen spezielle Layouts und Ressourcen bereitstellen.

Es gibt Situationen, in denen wir die automatische Anpassung des Koordinatensystems verhindern wollen, um unsere Anwendung immer im Querformat oder immer im Hochformat zu betreiben. Das ist unter anderem bei Spielen sinnvoll, die sehr häufig eine Spielfläche im Querformat besitzen, wie klassische Jump-and-Run-Spiele oder Spiele die den Ausblick aus einem Cockpit erlauben. Ein anderes Beispiel ist eine E-Book-Applikation, die das liegende Lesen ermöglichen soll und die Buchdarstellung im Hochformat fixiert, damit sich das Ding zumindest ein bisschen wie ein echtes Buch verhält.

TIPP

Wir sollten alle Dinge, die mit Sensoren und der Kamera oder anderen lageabhängigen Elementen zu tun haben, möglichst so gestalten, dass sie mit der automatischen Lageanpassung zurecht kommen und mit Hochformat- bzw. Querformatgeräten gleichermaßen korrekt arbeiten. Wenn das 100%ig funktioniert, steht auch dem festen Betrieb in Hoch- oder Querformat nichts im Wege. Dabei sollten wir uns immer fragen, ob das wirklich notwendig ist oder die Freiheit des Anwenders einschränken würde.

Sowohl wenn die automatische Lagenanpassung stattfindet als auch bei fixierter Lage gibt das Gerät über `getWindowManager().getDefaultDisplay().getRotation()` immer Aufschluss darüber, wie das Gerät gegenüber seiner natürlichen Lage gedreht wurde. Der Wert, den dieser Aufruf liefert, bestimmt immer, wie das Benutzerinterface gekippt werden musste/hätte gekippt werden müssen, um wieder korrekt dargestellt zu werden. Drehen wir ein Gerät um 90° nach **links**, so muss das User-Interface um 90° nach **rechts** gekippt werden und der Aufruf liefert als Ergebnis `Surface.ROTATION_90`. Kippen wir das Gerät um 90° nach **rechts**, muss das Userinterface um 90° nach **links** gekippt werden und der Aufruf liefert 360° – 90° = `Surface.ROTATION_270`.

Wenn wir uns mit der Programmierung Schritt für Schritt beschäftigen werden wir uns ein paar Kniffe ansehen, um mit den unterschiedlichen Koordinatensystemen optimal umzugehen.

1.7.2 Smartphones und Tablets

Abhängig vom Formfaktor (äußere Abmessungen) des Geräts unterscheidet man zwischen Smartphones und Tablets. Bis zu einer Bildschirmdiagonalen von etwas über 4 Zoll kann man von einem Smartphone sprechen, wenn Telefoniefunktionalität angeboten wird, zwischen 5 Zoll und ca. 7 Zoll ist eine genaue Bezeichnung etwas schwieriger. Als Tablet würde ich diese Geräte noch nicht bezeichnen, als Smartphone aber wegen der Größe auch nicht mehr. Da allen Android-Geräten gemeinsam ist, dass sie mindestens eine Form der Netzwerkkonnektivität bereitstellen, könnte man alle diese Geräte als Mobile Internet Devices (Mobile Internetgeräte) klassifizieren. Ab 7 Zoll Bildschirmdiagonale ist die Bezeichnung Tablet schon geläufig und treffend. Ich persönlich empfinde aber eine Bildschirmdiagonale ab 9 Zoll, besser 10 Zoll, als Minimum, damit ich das Tablet gerne einsetze.

Android trägt den unterschiedlichen Bildschirmgrößen mit verschiedenen Konzepten Rechnung.

Bedienkonzepte

Mit Android 3 und in Verbindung mit den Tablet-Formaten führt Android neue Bedienkonzepte ein, die auf kleinen Smartphones bisher nicht notwendig waren. Dazu gehört die Action Bar, die als Leiste am oberen Bildschirmrand den Zugriff auf wichtige Anwendungsfunktionen ständig präsent hält und z.B. Menüpunkte, aber auch Suchfelder und Navigationselemente aufnehmen kann.

Ein weiteres Element ist die System Bar, die am unteren Bildschirmrand die Standardnavigationstasten enthält, Benachrichtigungen aufnimmt und den direkten Zugriff auf die Geräteeinstellungen erlaubt.

In Verbindung mit der System Bar wurde die Recent-Apps-Liste eingeführt, in der die aktuell laufenden Anwendungen aufgelistet werden, um schnell zwischen den Anwendungen wechseln zu können.

Neue Interaktionsmöglichkeiten sind mit einem verbesserten Clipboard für Copy&Paste sowie mit einem neuen Drag&Drop-Framework eingeführt worden.

Im Bereich der App Widgets, das sind die kleinen Anwendungen die auf dem Homescreen herumlungern können, gibt es neue Funktionen, um Sammlungen wie Bilder als 3D-Stapel anzuzeigen und durchblättern zu können, und ab Android 3.1 können diese Widgets frei skaliert werden.

Konfigurationsabhängige Ressourcen

Ein zentraler Bestandteil um den untereschiedlichen Formfaktoren Rechnung zu tragen, ist das konfigurationsabhängige Ressourcensystem. Das Ressourcensystem ermöglicht es, für unterschiedliche Bildschirmgrößen, aber auch für unterschiedliche Ausrichtungen und Auflösungen angepasste Layouts bereitzustellen.

Styles und Themes

Auf dem Ressourcensystem aufbauend können wir Stilvorgaben und sogenannte Themes benutzen. In den Stilvorgaben können wir festlegen, ob z.B. die Action Bar sichtbar ist. In Verbindung mit den konfigurationsabhängigen Ressourcen können wir dann in Zukunft unsere Anwendungen damit für kleine und große Bildschirme konzipieren.

Systeminformationen

Das Framework bietet uns umfangreiche Methoden, um die Eigenschaften des Geräts abzufragen und in Erfahrung zu bringen, was es alles hat und kann. Damit können wir sicherstellen, dass die Anwendung mit unterschiedlichen Konfigurationen gut zurecht kommt.

In Verbindung mit den konfigurationsabhängigen Ressourcen haben wir damit alle Möglichkeiten, die wir für eine übergreifende Anwendungsentwicklung brauchen. Allerdings bedeutet es auch eine gewisse Planungsarbeit, und umfangreiche Tests auf verschiedenen Geräten sind angeraten.

Policies und das Manifest

Das Manifest jeder Android-Anwendung gibt dem System Auskunft darüber, was unsere Anwendung alles bereitstellt und auch was es an Systemvoraussetzungen benötigt. Hier können wir bereits sehr genau festlegen, auf welchen Geräten die Anwendung überhaupt

installiert werden kann und welche Berechtigungen der Anwender unserer Anwendung einräumen muss, um sie zu installieren.

Die grundlegende Eigenschaft, die wir im Manifest festlegen, ist, welche Betriebssystemversion wir mindestens voraussetzen. Darüber hinaus können noch Angaben zur benötigten Hardware festgelegt werden, und wenn wir auf sicherheitskritische Systemkomponenten wie das Netzwerk, Location Services oder die Telefonie und die Kontakte zugreifen wollen, müssen wir diese Zugriffe im Manifest deklarieren.

1.8 Zusammenfassung

Wir haben uns hier angeschaut, was das Betriebssystem Android und Android-Geräte alles bieten können. Durch das offene Konzept gelangen wir über die Android-Developer-Seiten an alle Informationen, die wir zum Entdecken von Android brauchen. Die Werkzeuge dafür sind alle im Rahmen von diversen Open-Source-Lizenzen verfügbar. Um nun das, was wir hier erfahren und besprochen haben, auch in konkrete Anwendungen umzusetzen, benötigen wir eine Entwicklungsumgebung und tieferen Einblick in die Programmierung. Im nächsten Kapitel wollen wir uns damit beschäftigen eine lauffähige Entwicklungsumgebung aufzubauen.

Auf geht's!

2 Einrichten der Entwicklungsumgebung



Wer immer tut, was er schon kann, bleibt immer das, was er schon ist.

Henry Ford, 30.07.1863 – 07.04.1947

Gründer von Ford

Ein guter Ausgangspunkt für alle Aktivitäten bezüglich Android ist die Website `http://www.android.com`. Von hier aus sind die Informationen für Partner und Entwickler sowie der Zugang zum Android-Market zu finden.

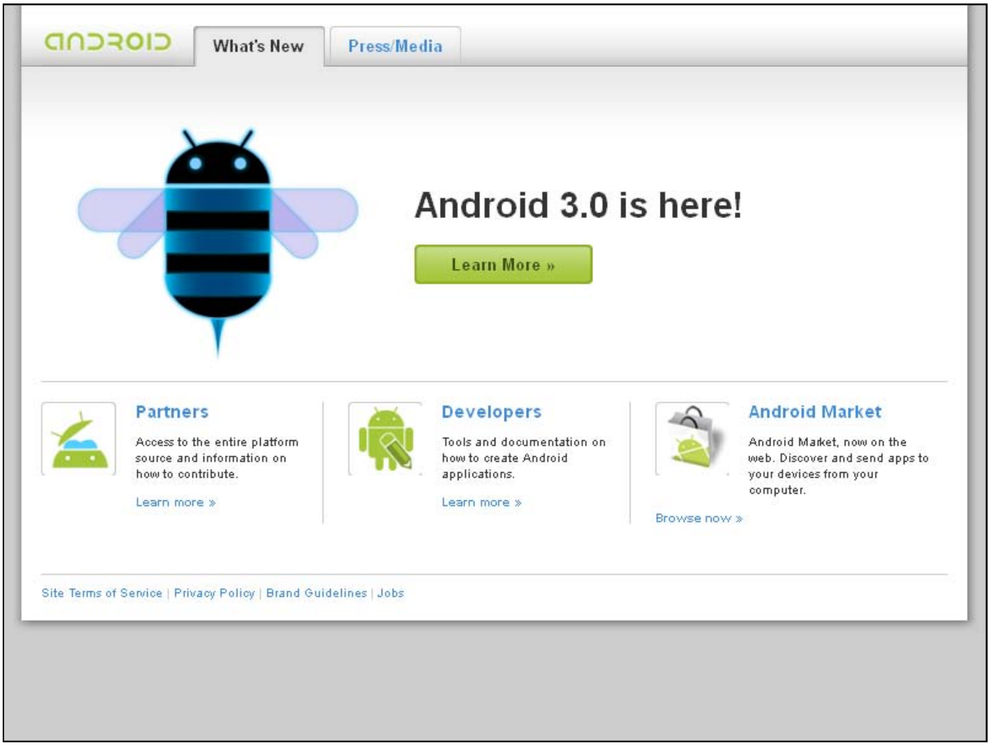


Abbildung 2.1: **www.android.com – wo alles beginnt …**

Der Link *Partners* führt zur Seite `http://source.android.com`. Wenn wir ein tolles neues Gerät entwickelt haben, ein Smartphone, ein (Web-)Tablett, einen neuartigen Radiowecker oder Kühlschrank, auf dem wir Android als Betriebssystem verwenden wollen, finden wir hier alles, was wir benötigen um Android auf unsere Hardware zu portieren, auf unsere Belange anzupassen, oder auch um neue Funktionalitäten und Verbesserungen zum Betriebssystem beizusteuern.

Das ist allerdings ein ganz anderes Thema und füllt für sich genommen locker ein bis mehrere weitere Bücher. Ich denke, ich werde mich damit beschäftigen, wenn ich hiermit fertig bin. Ich träume ja immer noch vom eigenen Gadget, das alles in den Schatten stellt was Apple und Konsorten jemals erdacht haben. Aber konzentrieren wir uns erst einmal auf die Entwicklung von Apps.

Das führt früher oder später zum Android-Market. Folgen wir also dem Link *Android Market* zur Seite `http://market.android.com`. Der Android-Market ist der Marktplatz, auf dem Hunderte, gar Tausende Applikationen für Android verschenkt oder verkauft werden. Zum Laden und Installieren stöbern wir üblicherweise mit der App *Market* unseres Android-Geräts im Android-Market. Die App ist wesentlich komfortabler als die Webseite, sie bietet z.B. eine Suchfunktion, mit der wir gezielt nach Applikationen suchen können. Allerdings gelangt man auf der Webseite über den Link *If you are a developer, learn about publishing your application here* dorthin, wo wir uns als *Publisher* anmelden können.

Mit dem Android-Market beschäftigen wir uns später an anderer Stelle, nämlich dann, wenn es darum geht, unsere eigenen Apps über diese Plattform anzubieten. Jetzt müssen wir aber erst einmal eine App entwickeln.

Also folgen wir dem Link *Developer* zur Seite http://developer.android.com. Jetzt sind wir an der richtigen Stelle, um loszulegen. Hier finden wir alles, was das Herz begehrt und was wir benötigen, um eine lauffähige Entwicklungsumgebung zur Entwicklung von Apps aufzubauen.

TIPP

Regelmäßiges Stöbern auf der Seite lohnt sich, sowohl im Abschnitt *What's New* als auch unter *Press/Media*. Unter *What's New* werden im *Android Developers Blog* immer wieder interessante Informationen, Tipps und Best-Practice-Lösungen veröffentlicht, und unter *Press/Media* erhält man einen öffentlichkeitswirksamen Überblick über die Neuerungen bei Einführung neuer Versionen.

Die Basis zur Entwicklung von Anwendungen für das Betriebssystem Android ist das Android-SDK. Applikationen für Android werden ausnahmslos in Java erstellt, sodass im Prinzip jede beliebige Java-Entwicklungsumgebung zum Einsatz kommen kann. Zwar befindet sich in Android keine Java-Virtual-Machine, dafür aber die Dalvik-Virtual-Machine. Eine Art Cross-Assembler überführt den Java-Bytecode der Java-Class-Files (`.class`) in den Bytecode der Dalvik-VM (Dalvik Executable, `.dex`). Dabei werden die Class-Files zu einem DEX-File zusammengefasst und weitere Optimierungen durchgeführt.

Das Android-SDK liefert neben den spezifischen Android-Bibliotheken auch Portierungen der Java-Standardbibliotheken aus dem Apache Harmony-Projekt und einige weitere Apache-Bibliotheken mit.

Dieser technische Hintergrund ist insofern wichtig als dass es bedeutet: Wir entwickeln zwar in Java, es kommt aber kein Java zur Ausführung.

Eine Folge ist, dass die Android-Java-Klassenbibliothek nicht dem Java-Standard folgt und nicht mit Java ME (Micro-Edition, der SUN- bzw. Oracle Java-Standard für mobile Geräte) oder gar Java SE/EE (Standard-Edition/Enterprise-Edition, der Standard für Desktop-Anwendungen) kompatibel ist. Java ME-Applikationen, die z.B. auf Symbian-Geräten mit Java ME laufen, sind auf Android nicht lauffähig und können auch nicht so ohne Weiteres portiert werden, da die Java ME-Architektur nicht direkt auf die Dalvik-VM übertragen werden kann.

Möchte man zusätzliche Java-Bibliotheken benutzen, die nicht mit dem SDK mitgeliefert werden, so muss man darauf achten, dass

a) die CLASS-Files mit einem Original SUN-Java-Compiler übersetzt sind **oder**

b) wir die Quellen zur Verfügung haben **und**

c) wir keinen Namenskonflikt mit den Standardbibliotheken bekommen.

Bei der Verwendung externer Bibliotheken müssen wir weiterhin darauf achten, dass durch das ADT unsere gesamte Anwendung in ein *Android-Package* (*.apk*-Datei) gepackt wird und alle von uns verwendeten Ressourcen, Klassen und Bibliotheken dort hinein gepackt werden.

Im Folgenden betrachten wir, welche Voraussetzungen und Komponenten benötigt werden, danach schließt sich eine Schritt-für-Schritt-Installationsanleitung an.

2.1 Systemvoraussetzungen

2.1.1 Hardware und Betriebssystem

Das Android-SDK ist auf folgenden Plattformen verfügbar:

1. Windows XP, 32-Bit
2. Windows Vista, 32- und 64-Bit
3. Windows 7, 32- und 64-Bit
4. Mac OS X 10.5.8 oder höhere Version, aber nur auf Intel x86 basierenden Systemen
5. Linux (getestet auf Ubuntu Hardy Heron). Falls eine 64-Bit-Linux-Distribution verwendet wird, muss diese Distribution auch 32-Bit Anwendungen laufen lassen können.

Wie bei den meisten Entwicklungsmaschinen bedeutet ein Mehr an allem auch meistens ein Mehr an Komfort und Leistung. Je mehr Hauptspeicher und je besser der Prozessor, um so besser ist die Performance beim Übersetzen der Programme und beim anschließenden Ausführen im Emulator.

Die einzelnen Komponenten benötigen an Plattenplatz:

1. Android-SDK: ca. 1,1 GB
2. Java-Development-Kit: ca. 200 MB
3. Eclipse: ca. 150 MB (und mehr, abhängig von der Ausbaustufe)

Ich schreibe dieses Buch und entwickle die Beispiele z.B. auf einem DELL INSPIRION 9400 mit 2 GB Hauptspeicher, 1,7 GHz Prozessortaktfrequenz und 150 GB Festplatte. Als Betriebssystem verwende ich Windows 7 Ultimate, und die Bearbeitung geht mit dieser Ausstattung relativ flott von der Hand. Bemerkbar macht sich die im Vergleich zu Desktop-Systemen geringere Taktfrequenz und der etwas knappe Hauptspeicher allerdings beim Ausführen der Emulatoren. Der Verbrauch an Hauptspeicher steigt natürlich mit Verwendung der Eclipse und mit laufendem Emulator sowie weiterer offener Programme wie Browser, Grafikprogramm etc. ganz beträchtlich auf über 1 GB. Die Startzeit des Emulators bis zur Betriebsbereitschaft beträgt ca. 1,5 Minuten. Das Installieren einer Applikation auf dem Emulator dauert je nachdem dann auch noch mal bis zu 1,5 Minuten.

Also, spendiert euch mehr Hauptspeicher (4 GB) und einen schnellen Prozessor. Das hilft, die Anzahl der Kaffeepausen zu minimieren.

Sehr nützlich ist der Betrieb von zwei Bildschirmen. Ich lasse z.B. die Entwicklungsumgebung auf dem Hauptschirm laufen und den Emulator parallel auf dem zweiten Schirm. Außerdem habe ich auf dem zweiten Schirm immer den Browser und den Acrobat Reader offen, um mich durch die Dokumentation und interessante Webseiten zu wühlen.

2.1.2 Java JDK

Da die Quelle für die Überführung in den Dalvik-Bytecode tatsächlich Java-Class-Files sind, benötigen wir auf dem Entwicklungssystem ein Java-Development-Kit (JDK).

ACHTUNG

Es muss ein JDK verwendet werden. Eine Java-Runtime-Environment (JRE) reicht nicht aus.

Das JDK ist auf der Seite

`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

zu finden. Für alle, die sich wundern: Java ist eine innovative Erfindung von SUN. Die Firma Oracle hat vor nicht allzu langer Zeit die Firma SUN gekauft. Ich hätte es auch nie für möglich gehalten, aber der Markt ist eben ständig in Bewegung.

Ladet euch die Version 6 für euer Betriebssystem herunter und folgt der Installationsanleitung.

2.1.3 Entwicklungsumgebung

Es ist möglich, Applikationen auch ohne integrierte Entwicklungsumgebung zu schreiben, mit der Verwendung der Kommandozeilenwerkzeuge des Android-SDKs in Verbindung mit dem Build-Manager *Ant*. Ich persönlich finde dieses Vorgehen interessant, da man so einen tiefen Einblick in die Projektstruktur und die Funktionsweise der Tools erhält, allerdings ist es für einen schnellen Erfolg aufgrund der hohen Einarbeitungszeit eher kontraproduktiv.

Daher wenden wir uns direkt einer integrierten Entwicklungsumgebung (Integrated Development Environment – IDE) zu, die uns viel Arbeit abnimmt und es erlaubt, dass wir uns hauptsächlich auf den Entwicklungsprozess konzentrieren. Seit der Version 9 der Android-Development-Tools wird Eclipse ab der Version 3.5 unterstützt, die Version 3.4 nicht mehr.

Eclipse ist eine integrierte Entwicklungsumgebung, die ursprünglich von IBM für die Entwicklung von Java-Applikationen entwickelt wurde. 2001 wurde der Quellcode freigegeben und 2004 die Eclipse Foundation gegründet, die sich seither um die Weiterentwicklung kümmert. Heute ist Eclipse durch ein Plug-in-Konzept eine hochgradig flexible und erweiterbare Umgebung für Entwicklungsaufgaben aller Art.

Für Eclipse liefert das Android-SDK ein entsprechendes Plug-in, das die Erstellung und Konfiguration unserer Projekte übernimmt und uns von aufwendigen manuellen Konfigurationsschritten befreit. Außerdem erlaubt das Plug-in das Starten der Applikationen und die Fehlersuche entweder auf einem Emulator oder direkt auf einem Android-Gerät.

Alles rund um Eclipse finden wir auf den Seiten der Eclipse Foundation unter `http://www.eclipse.org`.

ACHTUNG

In der aktuellen Version der Android-Development-Tools wird Eclipse ab der Version 3.5 unterstützt, wir können also die aktuelle Version (Stand heute Eclipse 3.6) benutzen.

Die Version 3.4 wird nicht mehr unterstützt.

Nach dem wir uns für die richtige Versionsnummer entschieden haben gelangen wir zur eigentlichen Download-Seite. Wie beschrieben ist Eclipse eine flexible Umgebung, die mit Plug-ins für unterschiedlichste Aufgaben ausgestattet werden kann. Für einige Aufgabengebiete gibt es bereits vorgefertigte Installationspakete. Wir benötigen mindestens das Paket für Java-Entwickler *Eclipse IDE for Java Developers*, alternativ die *Eclipse Classic 3.6.x*.

TIPP

Eclipse ist meine bevorzugte Entwicklungsumgebung für Java-Projekte. Durch die Plug-in-Architektur lässt sich Eclipse um viele Werkzeuge erweitern, z.B. um XML-Editoren und um zig Projekttypen für unterschiedliche Programmiersprachen. Es lohnt sich auf jeden Fall, sich mit Eclipse auch über die Android-Anwendungsentwicklung hinaus zu beschäftigen.

2.1.4 Das Android-SDK und die Android Development Tools (ADT)

Das ***Android-SDK*** ist das Herzstück der Entwicklung von Applikationen für Android. Das SDK stellt die Klassenbibliotheken bereit und die benötigten Werkzeuge, um aus unserem Quelltext lauffähige Android-Applikationen zu erzeugen.

Die Werkzeuge des Android-SDK sind fast ausschließlich Kommandozeilen-Werkzeuge, die sich entweder manuell oder per Ant automatisiert ausführen lassen.

Für alle, die sich mit den Interna nicht herumschlagen möchten, werden die *Android Development Tools (ADT)* für Eclipse bereitgestellt.

Das ADT-Plug-in bietet eine optimale Integration des SDK in die Eclipse-Umgebung und erweitert Eclipse um die Möglichkeit, Android-Projekte zu erzeugen, die Benutzeroberfläche zu entwerfen, die Klassen zu entwickeln und die Applikation entweder im Emulator oder auf einem angeschlossenen Android-Gerät zu testen. Weiterhin ist der Signierungsprozess zum Veröffentlichen der Applikationen integriert.

Damit stellen die ADT den schnellsten und einfachsten Weg dar, mit der Entwicklung von Android-Applikationen zu starten.

Sowohl das SDK als auch die ADT findet ihr unter http://developer.android.com/sdk/index.html.

Das SDK selbst liegt als *ZIP-Datei* für Windows und Mac OS X (Intel) bzw. als *TGZ-Datei* für Linux (i386) vor. Stand heute ist in diesen Paketen das sogenannte Starter Package enthalten, das nur die benötigten Werkzeuge und eine Management-Anwendung beinhaltet, über die die gewünschten SDK-Komponenten installiert werden können.

Die ADT werden nicht direkt von der Webseite heruntergeladen sondern über den Software-Update-Manager der Eclipse installiert.

2.2 Installation der Entwicklungsumgebung Schritt für Schritt

Nachdem wir nun die notwendigen Komponenten und deren Bezugsquellen kennen gelernt haben gehen wir den Installations- und Einrichtungsprozess Schritt für Schritt durch. Die Installationsdateien, die zum jetzigen Zeitpunkt gültig sind, findet ihr auch auf der beiliegenden CD im Verzeichnis *Installationsdateien*.

Dort befinden sich allerdings nur die jetzt gerade gültigen Dateien, und da sich Software ja so fürchterlich schnell weiterentwickelt, lohnt sich immer mal ein Blick auf die aktuellen Download-Seiten, allen voran immer auf die Seite `http://developer.android.com/sdk/requirements.html`, auf der ihr Informationen darüber findet, welche Eclipse-Version unterstützt wird und welche die aktuellen Werkzeuge oder JDK-Versionen sind, die ihr benutzen könnt.

2.2.1 Herunterladen und installieren des JDK

Wir öffnen die Seite `http://www.oracle.com/technetwork/java/javase/downloads/index.html` und gelangen zur Download-Seite für die verschiedenen Java SE – Varianten. Stand Juli 2011 ist die aktuelle Java-Version die Version 6. Achtet darauf, dass ihr das JDK in der Version 6 herunterladet bzw. die Version, die zum jeweiligen Zeitpunkt für Android freigegeben ist. Sollte eine andere Java-Version aktuell sein, gelangt ihr über den Link *Previous Releases* zu den vorherigen Versionen.



Abbildung 2.2: **Download-Seite für das JDK**

Wir wählen den Button *Java Download* aus der Button-Leiste der Download-Seite und gelangen nun auf eine Seite, auf der die Zielplattform ausgewählt werden kann und wir uns mit dem License Agreement einverstanden erklären, wenn wir auf *Continue* klicken.

Aus der Drop-down-Box *Platform* wählen wir unsere Zielplattform aus, ich verwende *Windows* als Betriebssystem.

Es ist möglich, sich als Benutzer bei der Oracle-Plattform zu registrieren. Als registrierter Benutzer kann man einige weitere Vorteile genießen, um das JDK herunterzuladen und zu nutzen, ist eine Registrierung nicht unbedingt nötig.

Nach dem wir auf *Continue* geklickt haben, öffnet sich eine weitere Seite, die nun endlich den eigentlichen Download-Link beinhaltet. Durch Aktivieren des Links *jdk-6u22-windows-i586.exe* (in meinem Fall) startet endlich der Download des Installers. Wir speichern den Installer in einem beliebigen Verzeichnis, aus dem wir den Installer später aufrufen können.

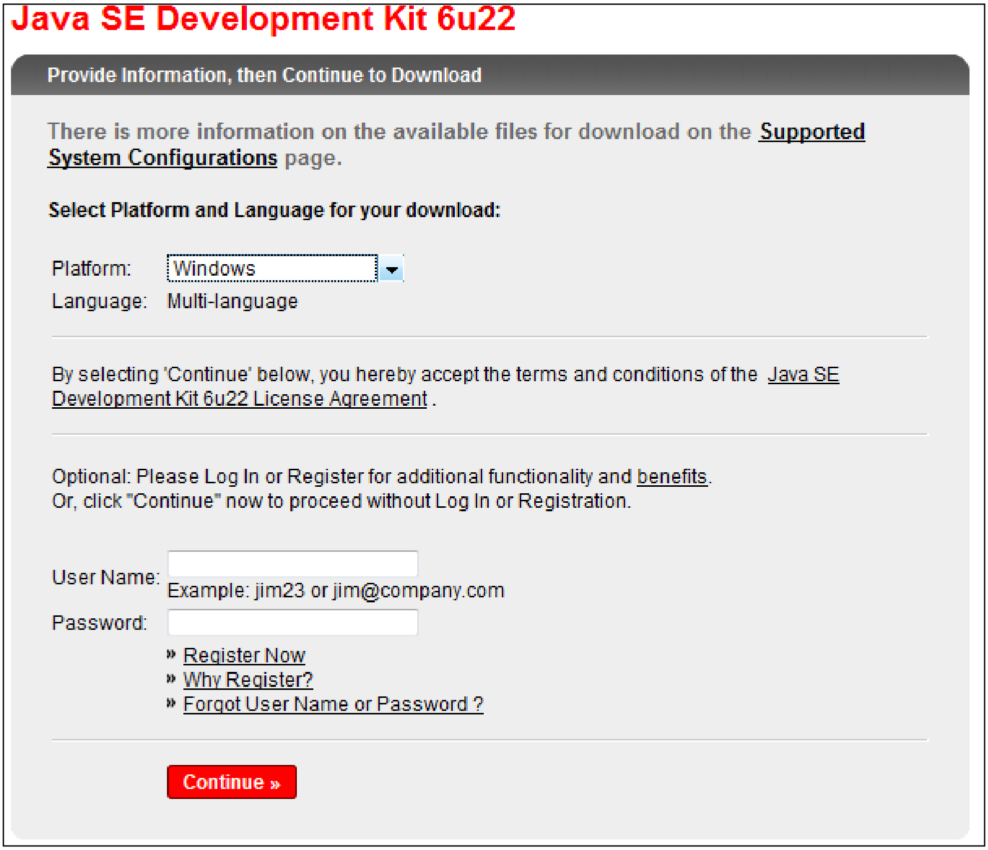


Abbildung 2.3: **Auswahl der Zielplattform und Registrierungsmöglichkeit**

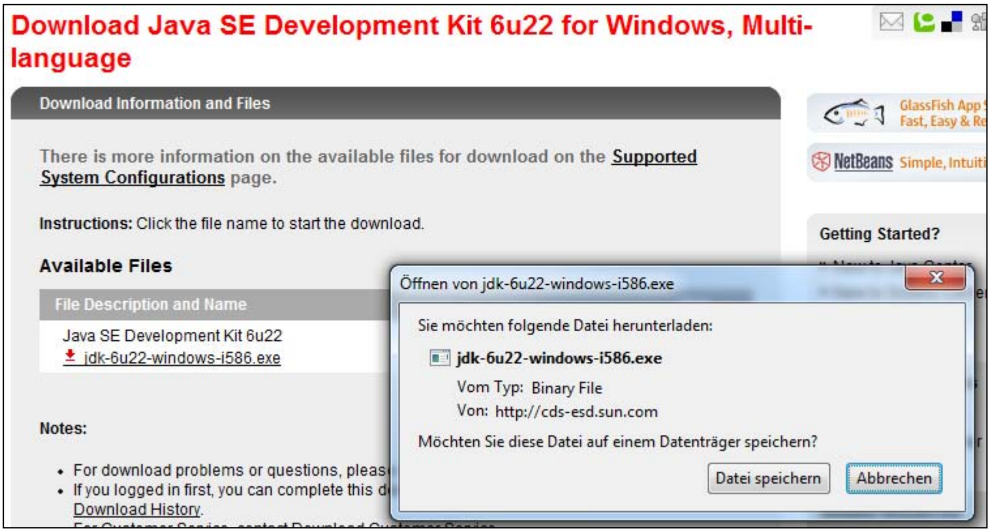


Abbildung 2.4: **Endlich – der eigentliche Download des JDK**

Wenn der Download abgeschlossen, ist rufen wir – unter Windows – den Installer auf und führen die Standardinstallation durch, indem wir einfach immer auf den Button *Next* klicken.

Ich habe die Standardeinstellungen beibehalten und auch das komplette Paket installiert. Wer ein wenig Platz sparen möchte, kann auf die Installation der Demos, des Source-Codes und der Java DB verzichten, denn diese Komponenten werden für die Android-Entwicklung nicht benötigt.

Nach dem letzten *Next* ist ein guter Zeitpunkt einen Kaffee zu trinken und die aktuellen Nachrichten zu lesen, die Installation dauert eine kleine Weile. Allerdings müssen wir zwischendurch nachschauen, denn wenn die JRE ebenfalls mit installiert wird (was sinnvoll für den reinen Betrieb der Eclipse ist), wird eine weitere Installation gestartet, die ebenfalls per *Next* durchgeklickt wird.

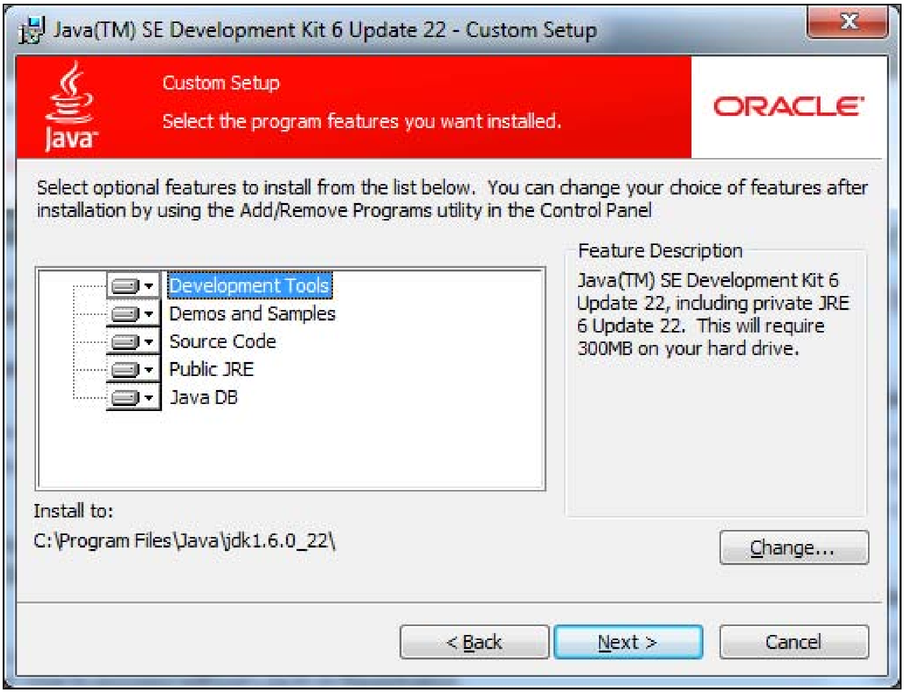


Abbildung 2.5: **Installation des JDK**

Nach Abschluss der Installation stellt uns Oracle frei, das Produkt zu registrieren. Die Registrierung verspricht die Versorgung mit aktuellen Informationen bezogen auf unsere Installation. Dafür möchte Oracle einige Informationen senden. Ich mag das nicht besonders und registriere mich aus diesem Grund meistens nicht, das ist aber jedem Anwender freigestellt. Wenn man die Registrierung nicht durchführt, bringt das keine Nachteile, und es werden auch keine Informationen an Oracle geschickt. Also bestätigen wir den Abschluss mit Finish, und jeder, der mag, ignoriert die Registrierungsseite die im Browser geöffnet wird.

2.2.2 Herunterladen und installieren des Android-SDK

Wir öffnen die Seite `http://developer.android.com/sdk/index.html`. Dort können wir das Android-SDK für unsere Plattform herunterladen. Seit der Version 9 der Development Tools findet sich für Windows eine Windows-Installer-EXE-Datei, mit der man die Tools installieren kann. Alternativ kann man die ZIP-Datei verwenden und die Tools aus der ZIP-Datei extrahieren.

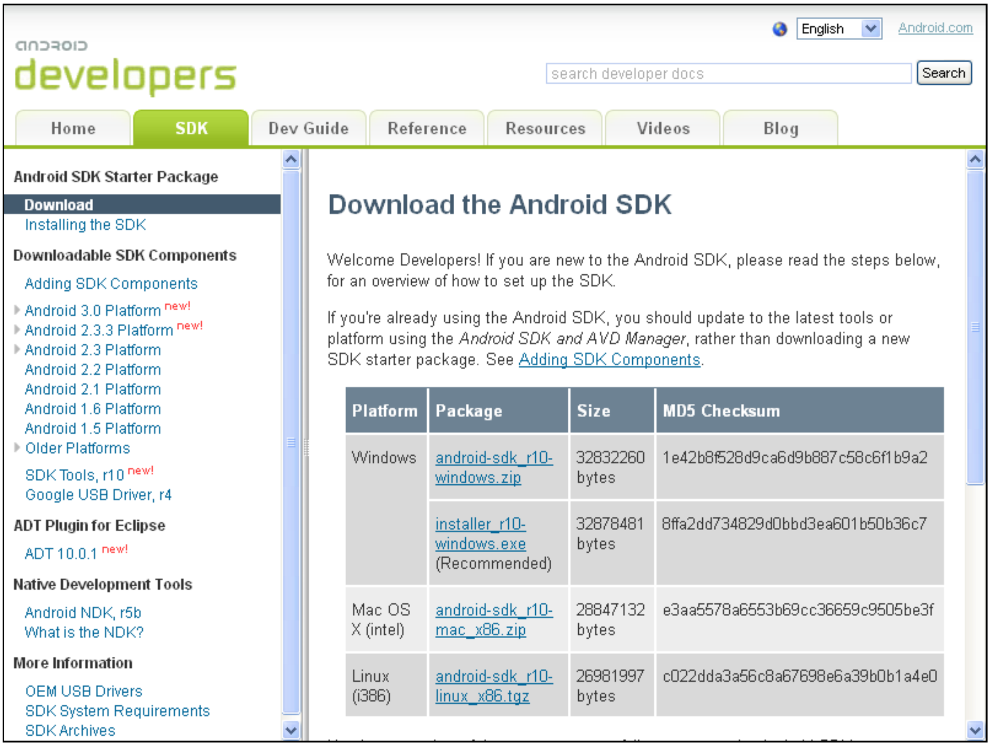


Abbildung 2.6: **Download-Seite des Android-SDK auf www.android.com**

Durch Aufruf des Installers starten wir die Installation. Als Installationsziel wird, wie unter Windows üblich, der Standardprogrammordner (`c:\Programme`, `c:\Program Files`) angegeben. Wohin auch immer wir die Tools installieren, wir müssen uns für die spätere Einrichtung des ADT-Plug-ins den Pfadnamen merken.

Es ist sinnvoll, die Tools in das Standardverzeichnis zu installieren.

Nach Abschluss der Installation kann man den SDK-Manager gleich starten, um die SDK-Komponenten nachzuladen.

Nach dem Start der Installation nimmt der Android-SDK and AVD Manager Verbindung zur Android-Seite unter dl-ssl.google.com auf, um die SDKs, die Beispiele und weitere benötigte Komponenten herunterzuladen. Je nachdem, für welche Android Version wir entwickeln wollen, können wir alle Packages herunterladen oder die Versionen abwählen, die wir nicht benötigen.

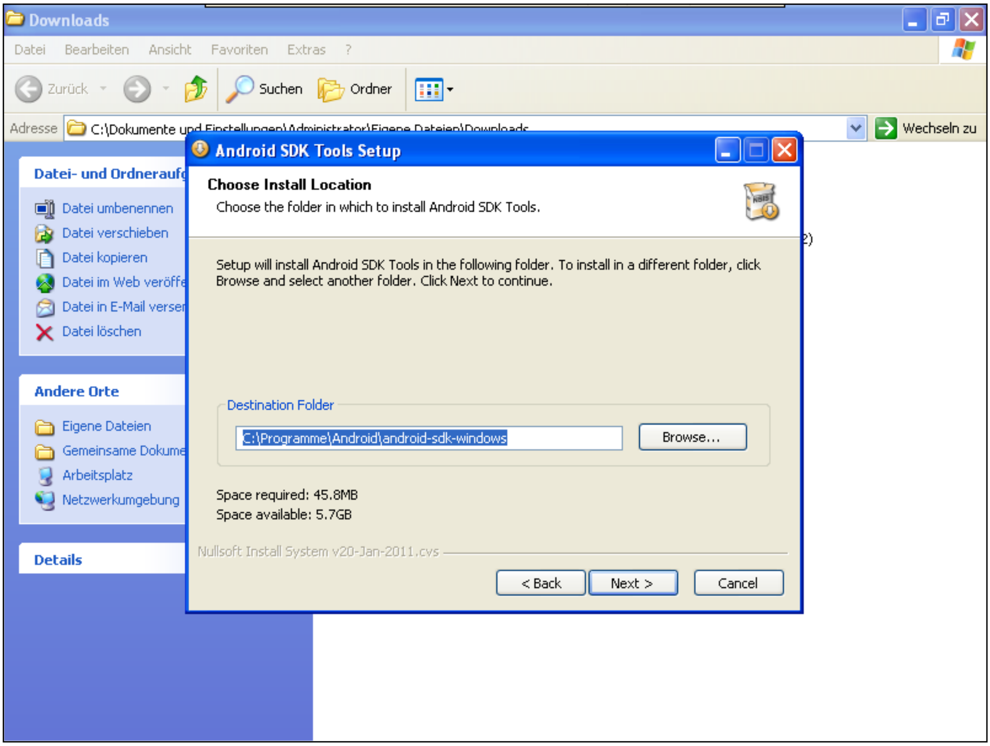


Abbildung 2.7: **Installation der SDK-Tools**

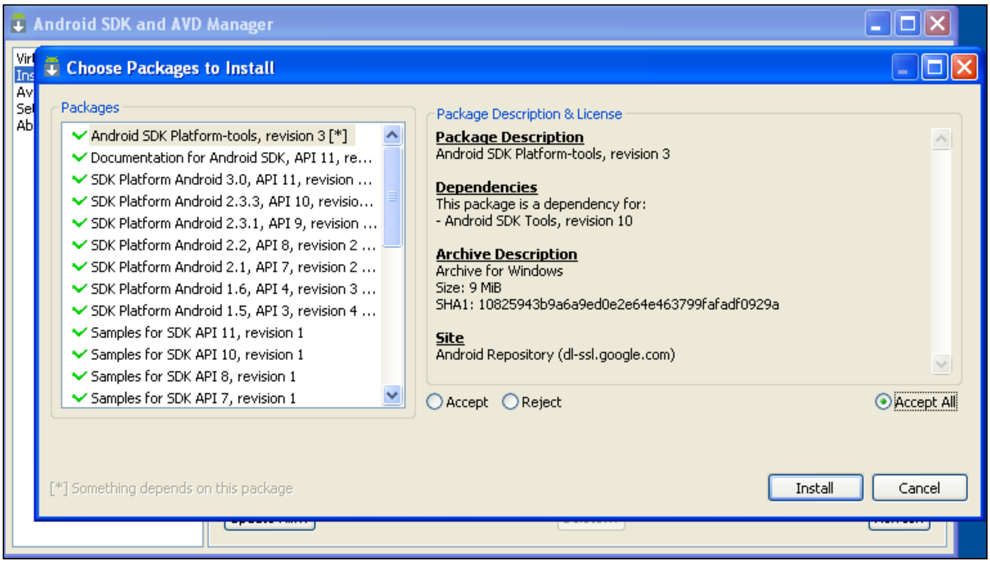


Abbildung 2.8: **Android-SDK and AVD Manager nach erstem Start**

Wichtig ist, die Option *Accept all* für die Package-Lizenzen anzuwählen, um alle Komponenten in einem Rutsch installieren zu können.

Nach Betätigen von *Install* werden die Komponenten heruntergeladen. Das kann einen ziemlich langen Moment dauern.

Nachdem die ausgewählten Tools, SDKs, Dokumentationen und Beispiele heruntergeladen sind, kann man innerhalb des SDK and AVD Managers nachschauen, welche Komponenten wir bereits installiert haben, und auch weitere optionale Bibliotheken und Werkzeuge herunterladen.

Aktualisierungen bereits heruntergeladener Komponenten können wir ebenfalls über den SDK and AVD Manager durchführen.

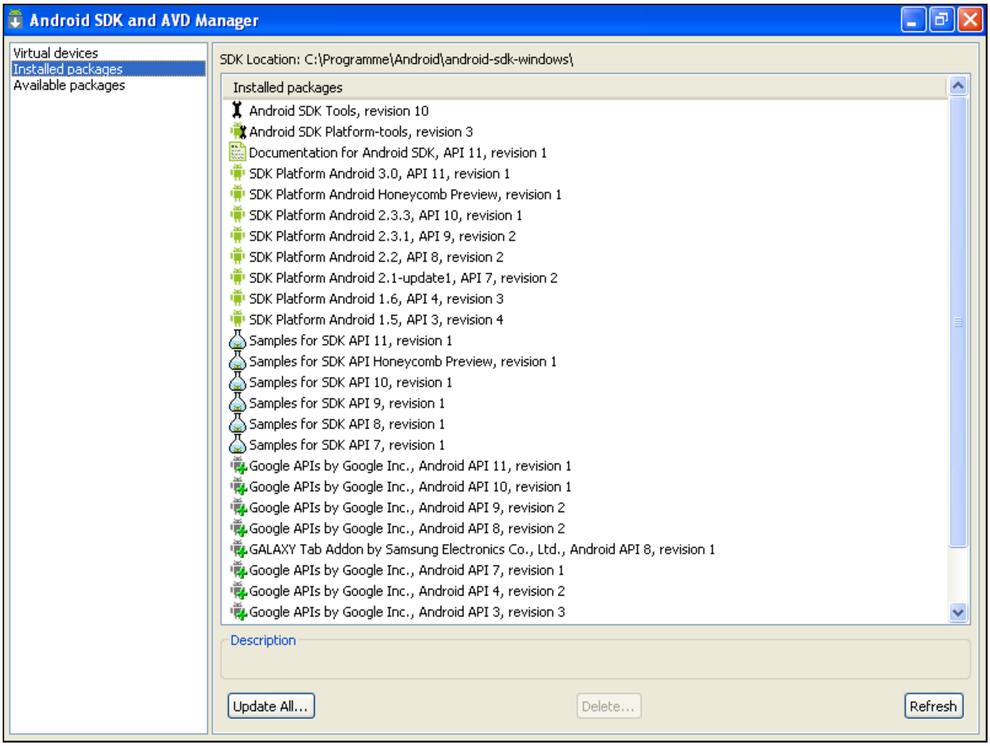


Abbildung 2.9: **Übersicht über die bereits installierten Komponenten**

Neben Google bieten einige Hersteller von Android-Geräten spezielle Bibliotheken an, um z.B. die Erweiterungen der Benutzeroberfläche auf dem Gerät in eigenen Programmen nutzen zu können oder spezielle Dienste des Herstellers anzusprechen.

ACHTUNG

Je spezifischer die Bibliotheken sind, umso geringer ist die Kompatibilität zu anderen Geräten. Wir sollten die speziellen Bibliotheken nur dann nutzen, wenn es entweder unumgänglich ist oder wir darauf achten, Alternativen für andere Geräte bereitzustellen.

Wichtige zusätzliche Komponenten können aber auch spezielle USB-Treiber sein, die wir zum Anschluss unserer Geräte an die *Android Debug Bridge* benötigen, um unsere Applikationen auf dem echten Gerät testen zu können.

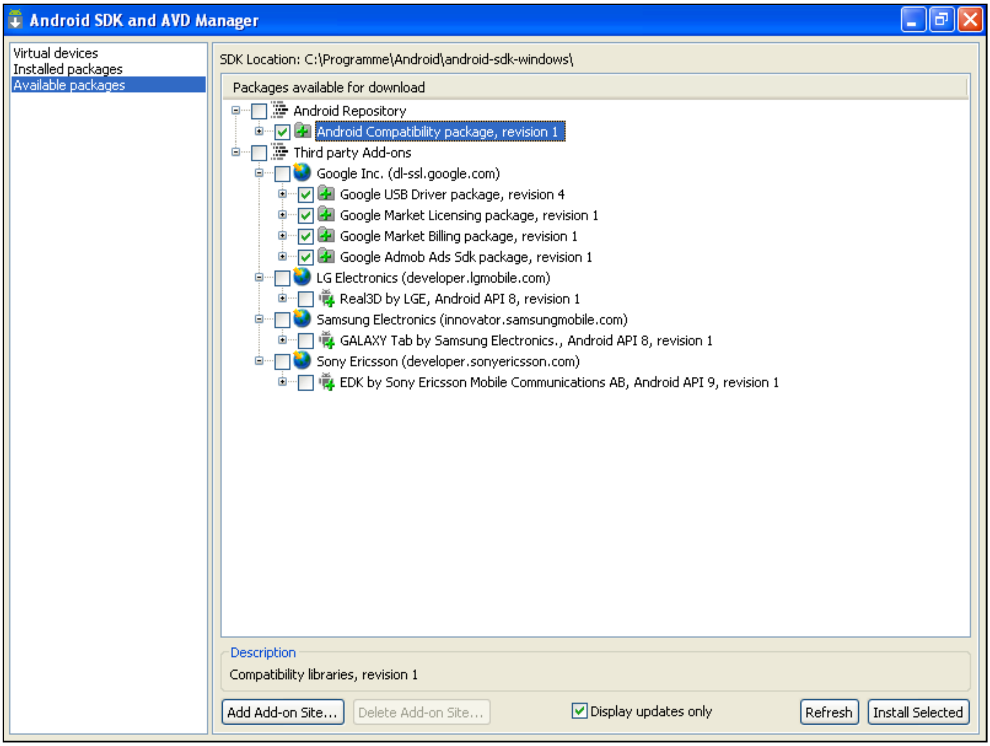


Abbildung 2.10: **Übersicht über die zusätzlich verfügbaren oder aktualisierten Komponenten**

Google selbst liefert z.B. Bibliotheken zum Zugriff auf Google Maps mit, in denen eine entsprechende Kartenansicht (*MapView*) und Zugriffsmethoden auf die Google Maps API zur Verfügung gestellt werden.

INFO

Bei den Bibliotheken von Drittherstellern versagt manchmal die Erkennung, ob die Komponente bereits installiert ist. Im Zweifel einfach in der Liste der installierten Komponenten nachschauen.

Nach dem Installieren könnten wir noch den Pfad zu den Werkzeugen des SDK, die unter dem Verzeichnis *c:\programme\android\android-sdk-windows\plattform-tools* zu finden sind, zu der Umgebungsvariablen PATH hinzufügen. Das ist dann nützlich, wenn wir die Werkzeuge direkt von der Kommandozeile aus benutzen möchten und nicht den kompletten Pfad angeben bzw. dorthin wechseln wollen. Da wir aber hauptsächlich die ADT aus der Eclipse heraus nutzen werden, ist das nicht unbedingt nötig.

2.2.3 Herunterladen und installieren der Eclipse

Wir öffnen die Seite `http://www.eclipse.org/downloads/` im Browser.

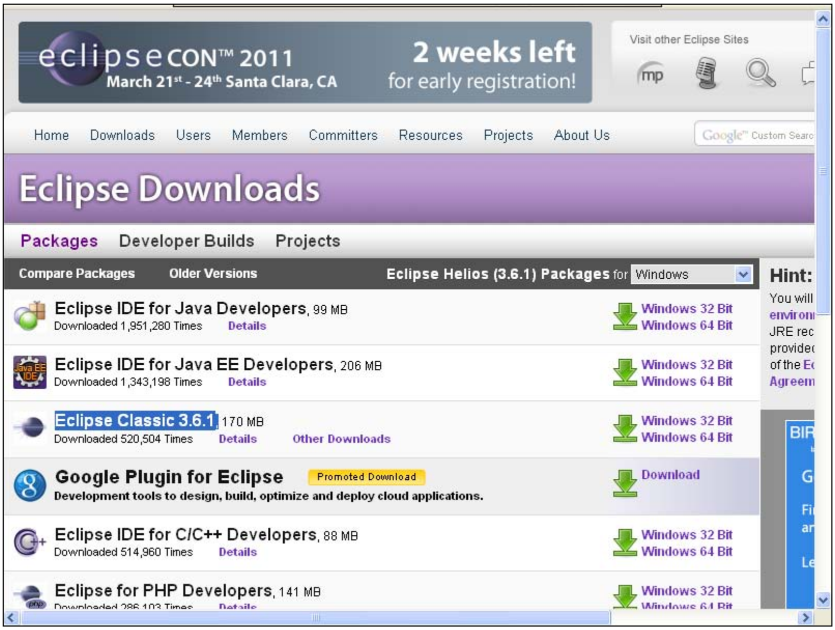


Abbildung 2.11: **Downloadseite für die Eclipse 3.6.1**

Stand April 2011 gelangen wir auf die Seite der Version *3.6.1 (Helios)*. Ich lade hier die Version Eclipse Classic 3.6.1 herunter. In der Classic-Edition sind bereits einige Plug-ins enthalten, die ich noch nützlich finde. Für die Android-Entwicklung reicht aber Eclipse-IDE for Java Developers aus.

INFO

Die Version 3.4 wird durch die neuen Android-SDK-Tools nicht mehr unterstützt. Wenn ihr diese Version bereits einsetzt, müsst ihr mindestens auf die Version 3.5 aktualisieren. Es ist aber auch möglich, mehrere Eclipse-Versionen parallel zu benutzen. Dazu muss beim Entpacken lediglich ein anderes Zielverzeichnis angegeben werden. Da Eclipse auf einen Installer und Einträge in die Windows-Registrierungsdatenbank verzichtet, ist der parallele Betrieb problemlos möglich.

Ich wähle die Version für *Windows 32 Bit*. Es öffnet sich die Download-Seite, auf der der *Mirror* (Spiegel) ausgewählt wird, von dem Eclipse heruntergeladen werden soll. Ich wähle eigentlich immer den vorgeschlagenen Mirror, es sei denn, ich stelle fest, dass der Download von dort sehr langsam läuft oder nicht richtig funktioniert. Nach dem Klick auf den Download-Link speichern wir die ZIP-Datei in einem entsprechenden Verzeichnis, aus dem wir die Eclipse-Umgebung dann entpacken.

Das Archiv besteht aus einem Hauptordner *eclipse*. Diesen Ordner entpacken wir in ein beliebiges Verzeichnis. Ich wähle dafür das Hauptverzeichnis *C:*, andere sinnvolle Verzeichnisse sind z.B. die Standard-Programmverzeichnisse (*C:\Programme* bzw. *C:\Program Files*).

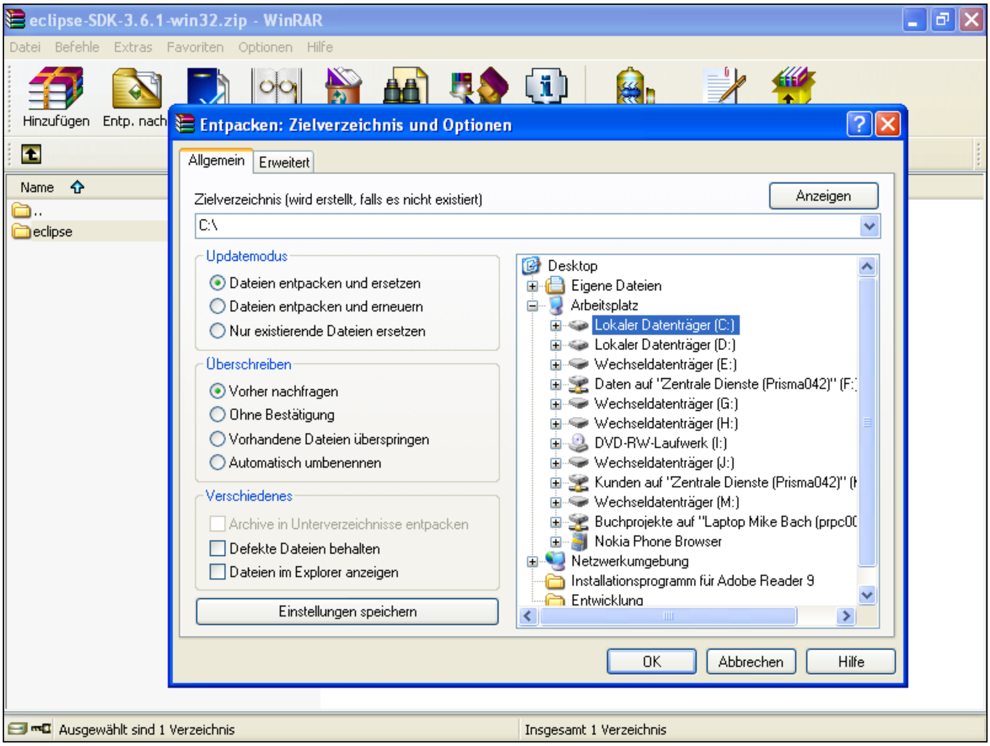


Abbildung 2.12: **Entpacken des ZIP-Archivs**

Nach dem Entpacken ist es sinnvoll, eine Verknüpfung auf dem Desktop oder in einem Verzeichnis des Startmenüs auf die Datei *c:\eclipse\eclipse.exe* bzw. der *eclipse.exe* im gewählten Pfad anzulegen.

INFO

Es ist problemlos möglich unterschiedliche Versionen der Eclipse zu installieren und parallel zu betreiben. Man muss einzig und allein ein anderes Zielverzeichnis zum Entpacken angeben. Es macht sicherlich Sinn, diese Verzeichnisse nach der enthaltenen Eclipse-Version zu benennen, z.B. *c:\Eclipse Helios* oder *c:\Eclipse 3.6*.

2.2.4 Erster Aufruf von Eclipse

Eclipse ist nun installiert. Mangels Installationsprogramm sind keine Verknüpfungen im Startmenü angelegt, es bleibt also uns überlassen wo wir uns eine Verknüpfung zur Eclipse.exe anlegen. Ich habe mir auf dem Desktop einen Ordner Entwicklung eingerichtet, in dem ich alle Verknüpfungen zu Entwicklungsumgebungen und Entwicklungswerkzeugen sammle.

Starten wir Eclipse nun zum ersten Mal (und auch bei allen weiteren Malen, wenn wir es nicht abstellen), fordert uns Eclipse auf, den *Workspace* auszuwählen.

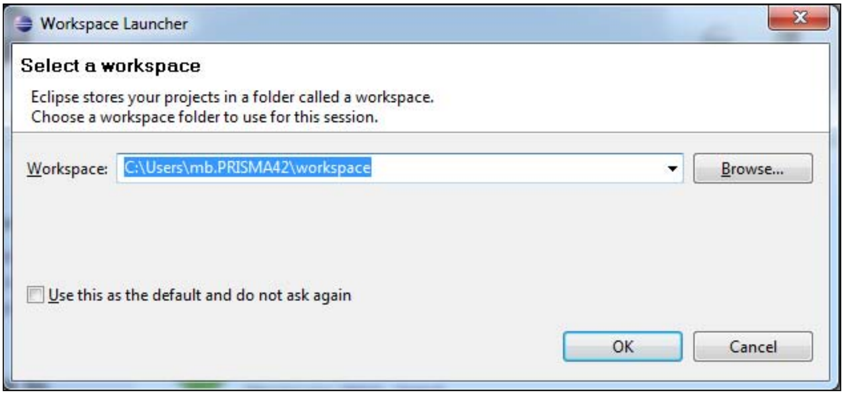


Abbildung 2.13: **Auswahl des Workspace**

Eclipse organisiert Projekte in sogenannten *Workspaces* (Arbeitsbereichen). Ein *Workspace* ist ein Verzeichnis auf der Festplatte und kann an einer beliebigen Stelle liegen.

Als Vorgabe schlägt Eclipse einen benutzerbezogenen *Workspace* vor, der im Heimatverzeichnis des Benutzers liegt, z.B. unter Windows 7: *C:\Users\<Benutzername>\workspace*.

Auf der CD ist ein *Workspace* enthalten, der die Projekte beinhaltet, die wir in diesem Buch erarbeiten werden. Um einen schnellen Überblick zu erhalten und direkt in den vorbereiteten Beispielen zu stöbern, kann dieser *Workspace* einfach auf die Platte kopiert und beim Start von Eclipse als *Workspace* ausgewählt werden. Um die Projekte Schritt für Schritt zu erarbeiten, empfiehlt es sich, entweder den vorgeschlagenen benutzerbezogenen *Workspace* zu verwenden oder einen leeren *Workspace* zu erstellen und zu benutzen.

Einen leeren *Workspace* erstellen wir, indem wir einfach ein neues Verzeichnis an einer beliebigen Stelle auf der Festplatte erstellen bzw. beim Start der Eclipse ein neues Verzeichnis im Startdialog benennen.

Zwischen unterschiedlichen Arbeitsbereichen kann später gewechselt werden. Entweder wählt man den *Workspace* jedes Mal beim Start von Eclipse aus, oder man legt einen *Workspace* als Vorgabe (*Use this as the default and do not ask again*) fest und kann die Bereiche über das Menü später wechseln.

Workspaces sind eine schöne Sache, wenn man ganz unterschiedliche Projekte (Android, Java, Java EE etc.) hat und diese voneinander trennen möchte. Und sie sind eine gute Organisationsmöglichkeit für große Projekte, die aus verschiedenen Entwicklungsprojekten, ggf. auch unterschiedlichen Typs, bestehen.

Nach Auswahl des Workspace startet Eclipse. Beim ersten Start zeigt Eclipse den Startbildschirm *Welcome*, über den ihr verschiedene Auswahlmöglichkeiten habt, z.B. herauszufinden, welche Features Eclipse bietet, Beispiele durchzuforsten, ein Tutorial zu bearbeiten oder aber zur *Workbench* (*Werkbank*) zu wechseln. Die *Workbench* ist die Umgebung, in der wir unsere Projekte erstellen, bearbeiten, ausführen und testen. Wechseln wir also zur *Workbench* und bereiten die Werkbank für den Einsatz vor.

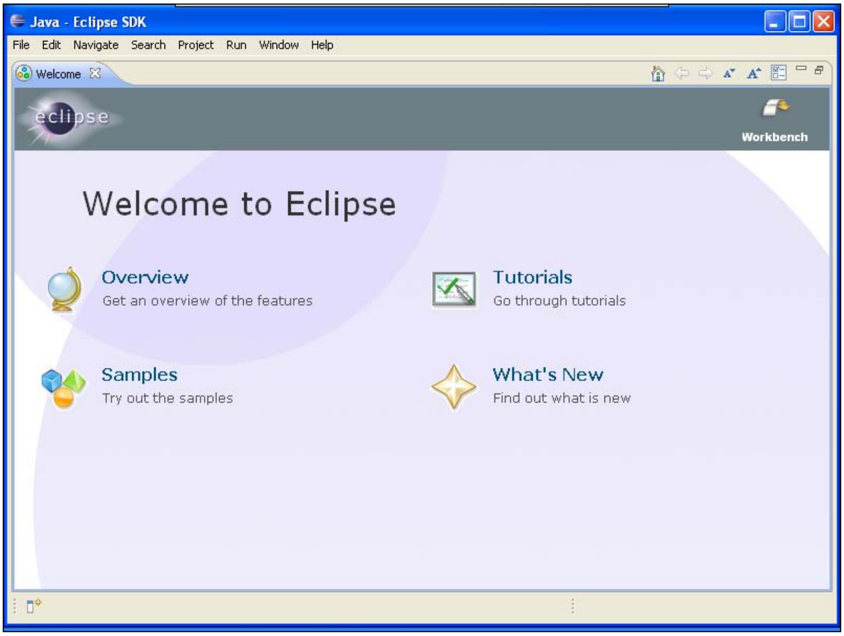


Abbildung 2.14: **Erster Start der Eclipse: Welcome!**

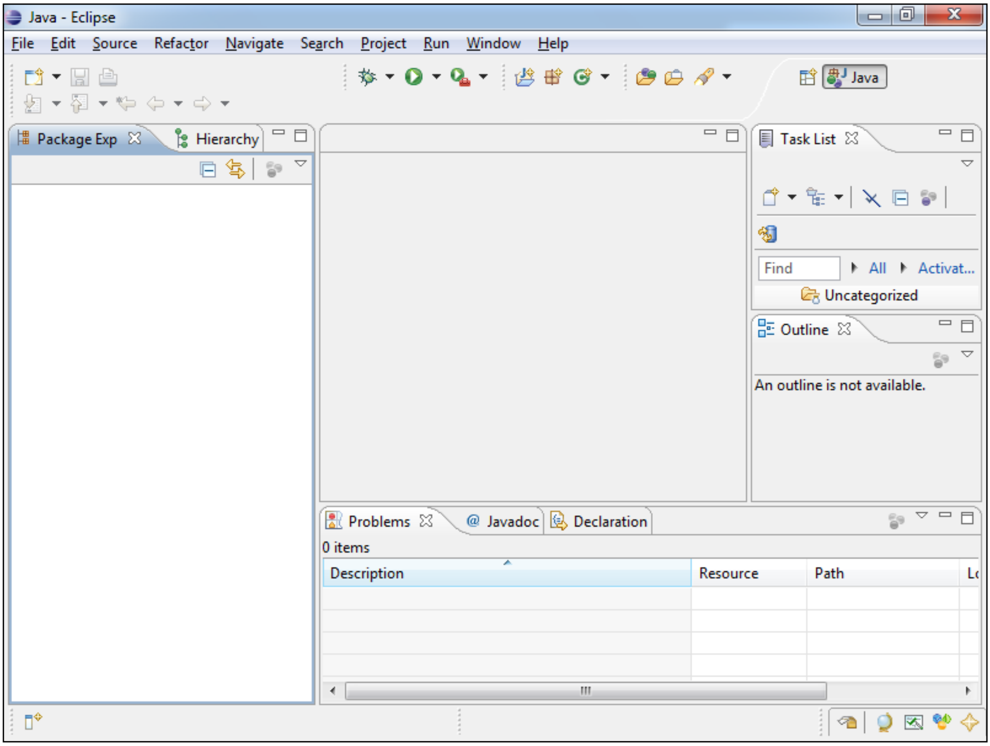


Abbildung 2.15: **Die Workbench**

2.2.5 Installieren des ADT-Plug-ins

Eclipse hat sich im Laufe der Zeit zu einer extrem flexiblen Entwicklungsumgebung für unterschiedlichste Aufgaben entwickelt, angefangen von der einfachen Java-Entwicklungsumgebung über die Unterstützung von komplexen Java-Enterprise-Projekten (Unternehmensanwendungen) bis hin zur modellgetriebenen Entwicklungsumgebung und vieles mehr. Die Flexibilität wird über ein Plug-in-Konzept erreicht das es erlaubt, die Entwicklungsumgebung modular zu erweitern, also weitere Editoren (z.B. grafische Editoren), weitere Compiler und andere Komponenten dem Grundsystem hinzuzufügen und so die Funktionalität beträchtlich zu erweitern. Mit dem Konzept kann man die Umgebung genauestens auf seine Anforderungen abstimmen, und es gibt zahlreiche vorgefertigte Plug-in-Pakete für viele Aufgaben.

Die Installation von Plug-ins kann manuell erfolgen, in dem ein Plug-in (die in der Regel als JAR Dateien vorliegen) einfach in das Plugin-Verzeichnis unterhalb des Eclipse-Programmverzeichnisses kopiert wird.

Wesentlich eleganter ist aber die Installation über den *Software-Update-Manager*, der in Eclipse integriert ist. Die meisten Plug-ins werden im Internet unter einer bestimmten Adresse als Pakete vorgehalten, die über den Update-Manager installiert werden können. Der große Vorteil ist, dass man über den Update-Manager auch bequem nach den aktuellen Versionen der installierten Plug-ins suchen und das System automatisch aktualisieren lassen kann. Ein weiterer Vorteil ist, dass bei bestehenden Abhängigkeiten zwischen Plug-ins diese Abhängigkeiten automatisch durch den Update-Manager aufgelöst und die benötigten weiteren Bestandteile automatisch nachgeladen werden können.

Auch das ADT-Plug-in, das für uns in diesem Zusammenhang relevant ist, lässt sich über den Software-Update-Manager installieren.

Der Software-Update-Manager findet sich unter dem Menüpunkt *Help*. Es gibt die zwei Menüpunkte *Check for Updates* und *Install New Software*. Um das ADT zu installieren, wählen wir *Install New Software* und gelangen in den Dialog *Install*.

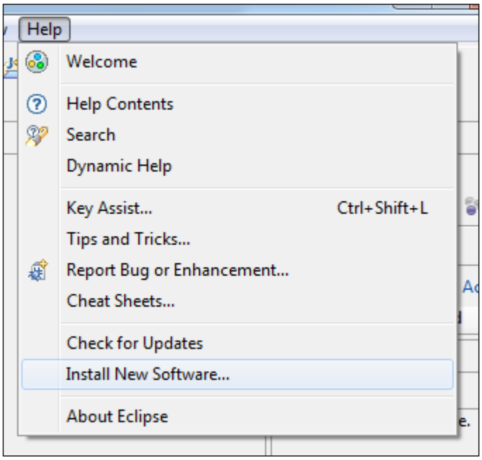


Abbildung 2.16: **Auswahl zum Installieren neuer Plug-ins**

Wir betätigen den Knopf *Add*, um das ADT-Plug-in zum Update-Manager hinzuzufügen. Im folgenden Dialog geben wir in das Feld *Name* einen Namen für die Update-Site ein, hier wählen wir den Namen *Android Plugin*. Unter diesem Namen erscheint die Update-Site später im Update-Manager. Im Feld *Location* geben wir die URL des ADT-Plug-ins `https://dl-ssl.google.com/android/eclipse/` ein und bestätigen den Dialog mit *OK*.

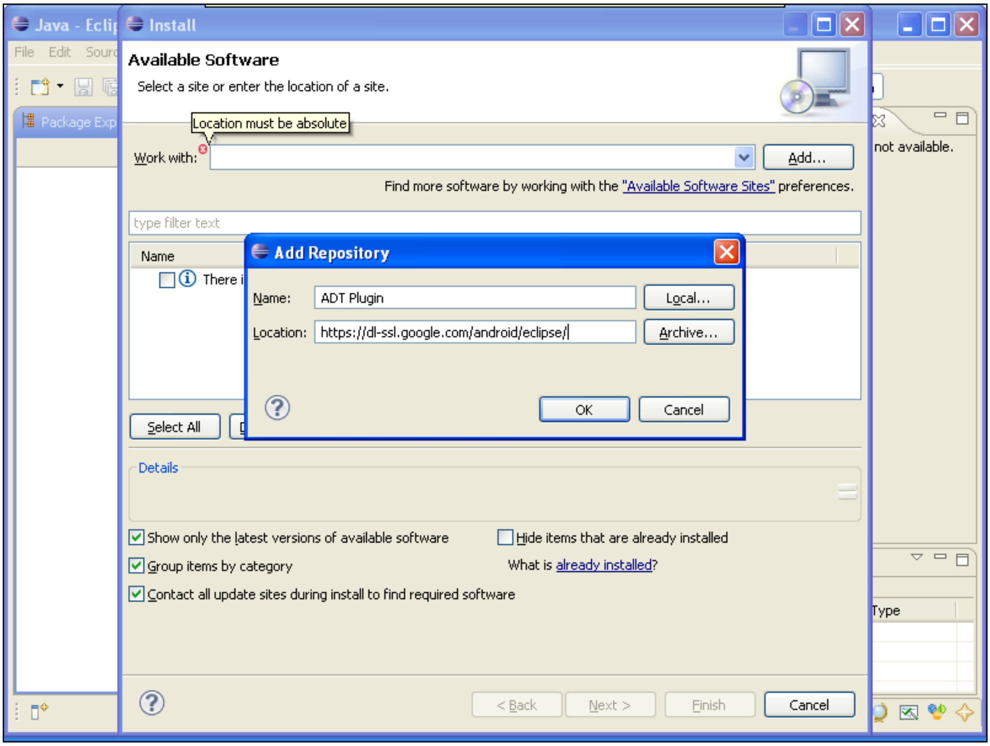


Abbildung 2.17: **Hinzufügen des ADT-Plug-ins**

Nach einiger Zeit sollte im Dialog in der Übersicht der Eintrag *Developer Tools* erscheinen. Sollte es dabei Probleme geben kann alternativ die URL `http://dl-ssl.google.com/android/eclipse/` verwendet werden (http:// statt https://).

Den Eintrag *Developer Tools* haken wir an und bestätigen mit dem Knopf *Next >*. Nach kurzer Zeit sollte im Dialog die Seite *Install Details* erscheinen, die uns die zu installierenden Komponenten anzeigt. Nach der Auswahl von *Next >* gelangen wir zur Übersicht der License Agreements, die bestätigt werden müssen. Die Option *I accept the terms...* wirkt sich direkt auf **alle** in der Liste aufgeführten Komponenten aus. Nun können wir mit *Finish* die Installation des Plug-ins endlich ausführen.

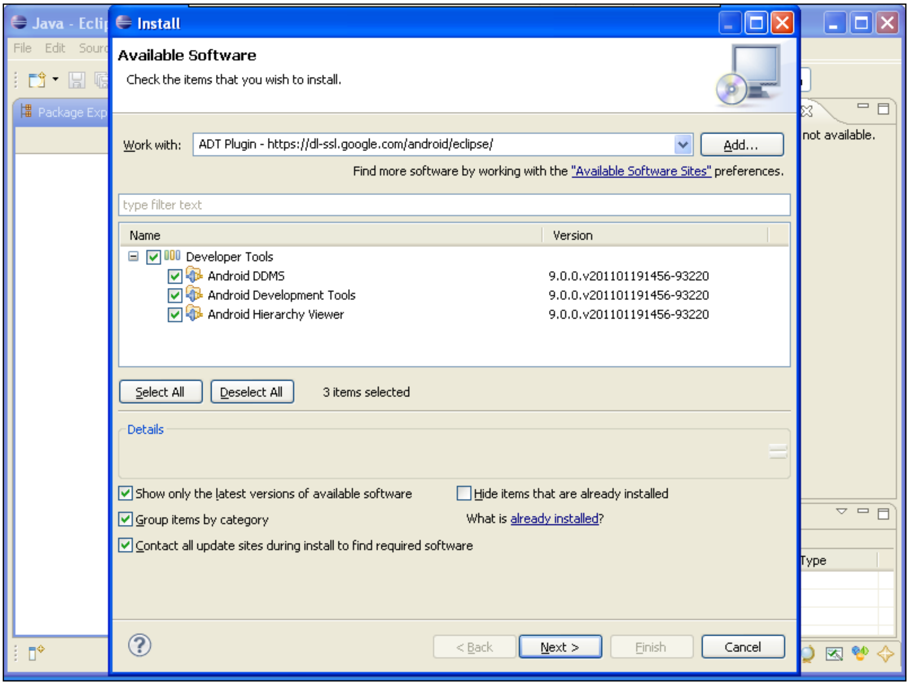


Abbildung 2.18: **Übersicht über die zu installierenden Komponenten (oberster Eintrag geöffnet)**

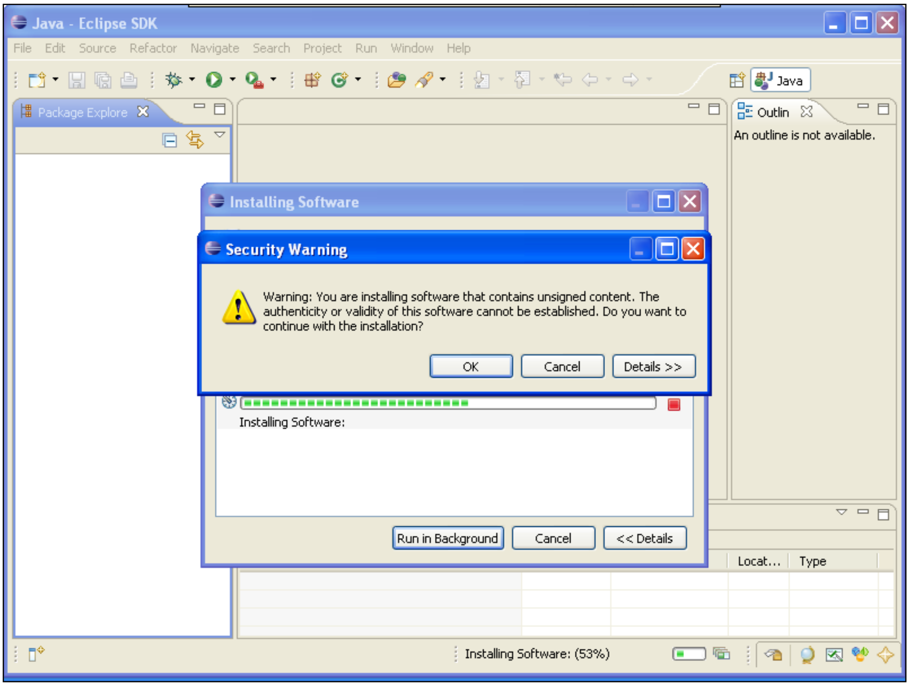


Abbildung 2.19: **Warnung während des Installationsprozesses**

INFO

Die Installation dauert ihre Zeit, Eclipse zeigt den Fortschritt an. Während des Installationsprozesses warnt Eclipse davor, dass wir *Software including unsigned content* installieren wollen. Diese Warnung können wir mit OK bestätigen, es handelt sich tatsächlich um die Android-Pakete von Google.

Nach Abschluss der Installation ist es empfehlenswert, Eclipse neu zu starten, dann können wir mit der Konfiguration des Plug-ins fortfahren.

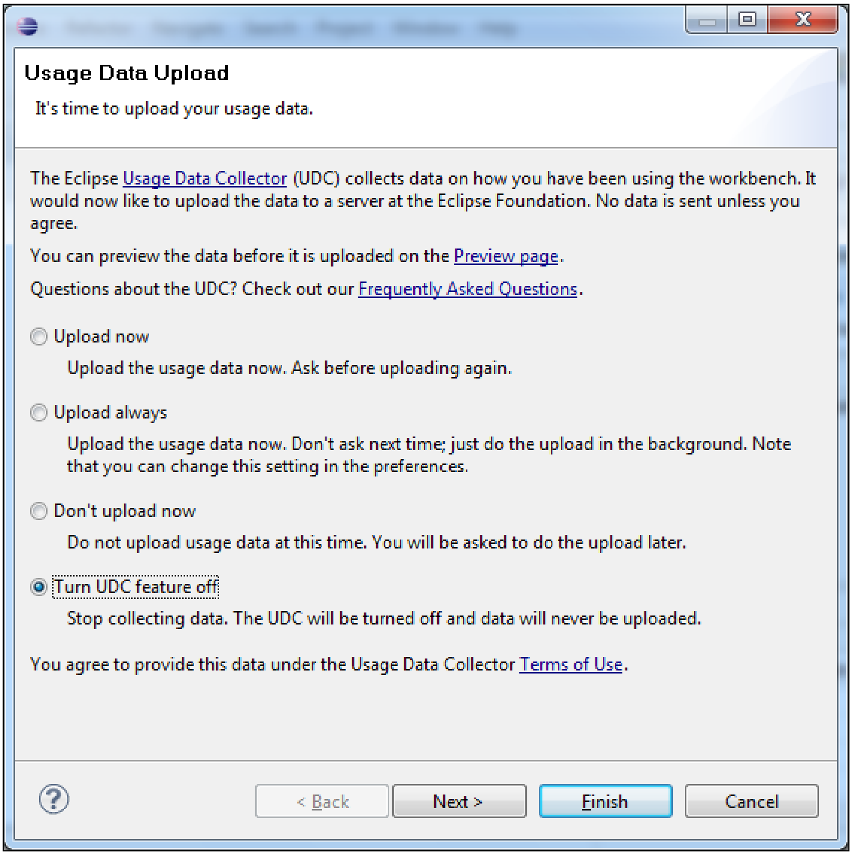


Abbildung 2.20: **Drolliger Versuch mein Nutzungsverhalten »auszuspähen«. Wofür eigentlich?**

Nach dem Neustart werden wir ggf. nach dem *Usage Data Upload* gefragt. Eclipse möchte gerne Statistiken über die Nutzung der Workbench anfertigen und benötigt dafür Nutzungsdaten. Das ist wiederum etwas, was ich nicht möchte und ich schalte das *UDC Feature* einfach aus. Ich weiß nicht, ob ich mich damit unsozial verhalte, aber das Datensammeln der Software, die auf meinem System installiert ist kommt mir immer komisch vor. Das machen ja in der Zwischenzeit einige Programme wie Microsoft Office und Microsoft Visual Studio so, aber auch viele Open-Source-Programme wie Open Office und Mozilla Firefox wollen manchmal Daten sammeln und nach Hause schicken. Mir ist bis heute schleierhaft, welchen Vorteil die Gemeinschaft der Nutzer davon hat, so richtig erklärt wird es einem

auch nicht. Ich habe schon früher immer herzlich gelacht über die Meldung nach einem Programmabsturz, dass man Microsoft oder den Hersteller benachrichtigen solle. Das Kürzeste vom Längsten: Ich schalte es aus. Was macht ihr?

2.2.6 Konfigurieren des ADT-Plug-ins

Nach dem Neustart hat sich augenscheinlich noch nicht viel verändert bis auf ein kleines Android-Icon in der Werkzeugleiste. Bevor wir das aber nutzen können, müssen wir dem Plug-in noch den Installationspfad des SDK- und AVD-Managers mitteilen.

Über den Menüpunkt *Window* → *Preferences* gelangen wir zum Konfigurationsdialog, mit dem alle Eclipse-Komponenten eingerichtet werden können. Unter dem Punkt *Android* finden wir das Feld *SDK Location*. Hier wählen wir den Pfad zum SDK aus, in meinem Fall ist das *c:\programme\android\android-sdk-windows*, und betätigen den Knopf *Apply*. Die Liste der *SDK Targets* wird dadurch aktualisiert, und wir sehen, welche SDK-Komponenten in der Basisinstallation vorhanden sind.

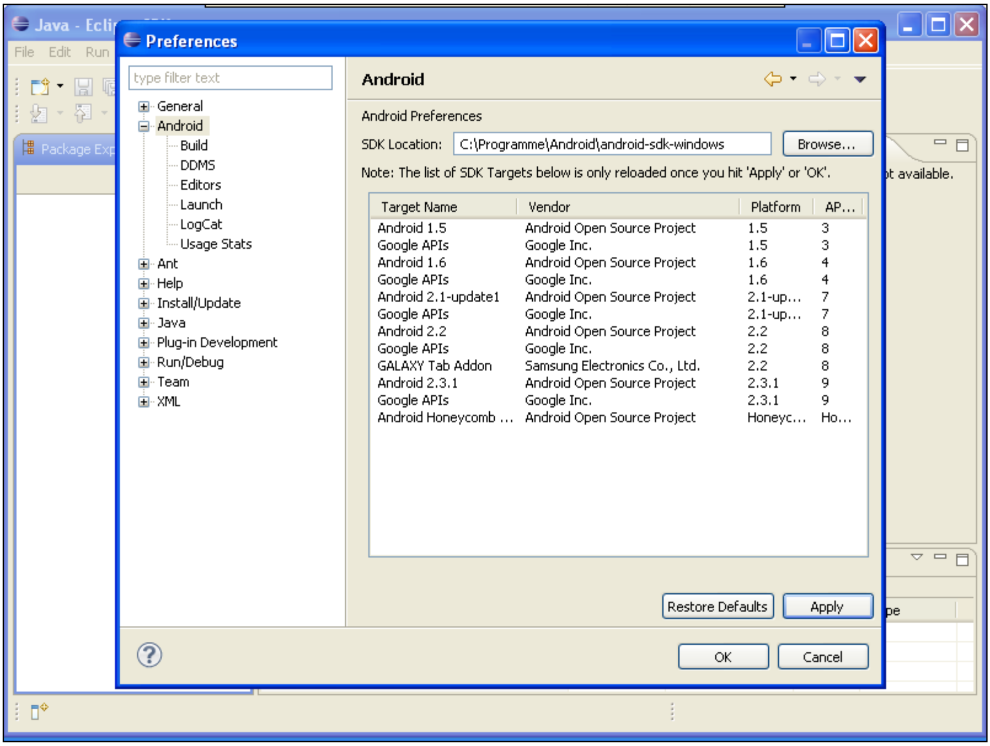


Abbildung 2.21: **Konfiguration des ADT-Plug-ins – nach Eingabe der SDK Location und Bestätigen per Apply**

Jetzt können wir die Konfiguration mit *OK* bestätigen. Mit diesem Schritt ist die Konfiguration erst einmal abgeschlossen, und wir können das ADT-Plug-in nutzen und endlich mit der Programmierung anfangen.

2.2.7 Aktualisieren des ADT-Plug-ins

INFO

Die nachfolgenden Schritte zeigen die grundsätzliche Vorgehensweise zur Aktualisierung unserer Entwicklungsumgebung bei neuen Versionen der Tools, neuer Betriebssystemversionen oder neuer Versionen des SDK-Managers. Die vorangegangene Beschreibung bezieht sich auf die Android-Version 3 und die Version 10 der SDK-Tools.

Wie das bei sehr aktiven Projekten üblich ist, kann es passieren, dass plötzlich Aktualisierungen vorhanden sind. So ist es mir auch beim Schreiben des Buches ergangen, hatte ich das ganze Thema zum Zeitpunkt von Android 2.2 begonnen zu bearbeiten wurde, Anfang Dezember die Version 2.3 und im Februar endlich die Verison 3 veröffentlicht – nachdem zwischendurch noch die Honeycomb-Preview erschienen war.

Die Veröffentlichung von neuen Versionen kann mehrere notwendige Aktivitäten nach sich ziehen:

1. Ggf. aktualisieren des Android-SDK Managers
2. Herunterladen des neuen SDK mit dem Android-SDK Manager

In Eclipse öffnen wir den Android-SDK Manager über den Menüpunkt *Window* → *Android-SDK and AVD Manager*.

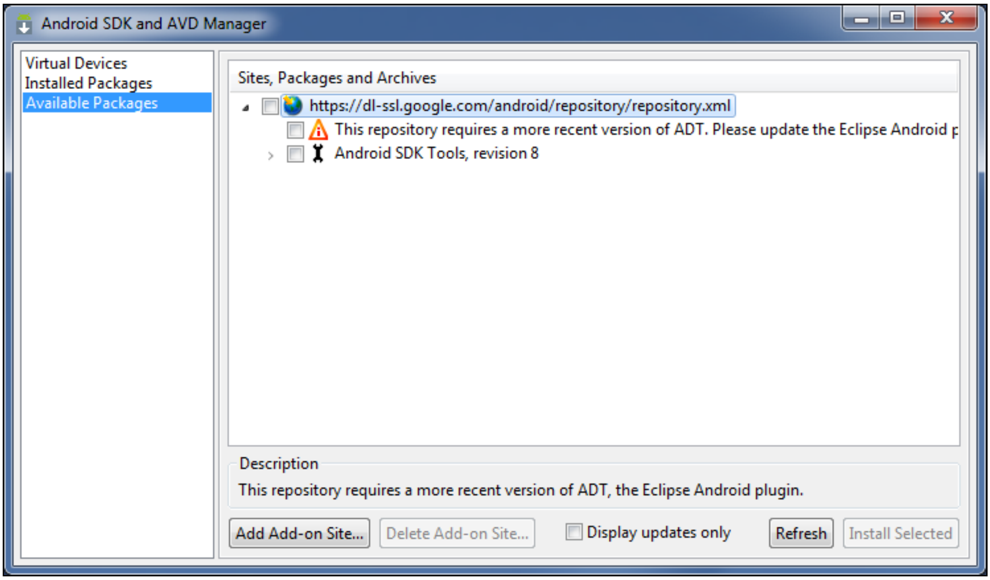


Abbildung 2.22: **Der Android-SDK and AVD Manager**

Um nach Aktualisierungen im Android Repository zu suchen, wechseln wir zur Option *Available Packages* und erweitern den Eintrag *https://dl-ssl.google.com/android/repository/repository.xml*. Darunter werden die verfügbaren neuen Packages angezeigt und, wie in diesem Fall, auch eine Warnung, dass das ADT-Plug-in aktualisiert werden muss.

Die Aktualisierung des ADT-Plug-ins erledigen wir mit dem Update-Manager der Eclipse. Der Update-Manager wird über *Help* → *Check for Updates* aufgerufen.

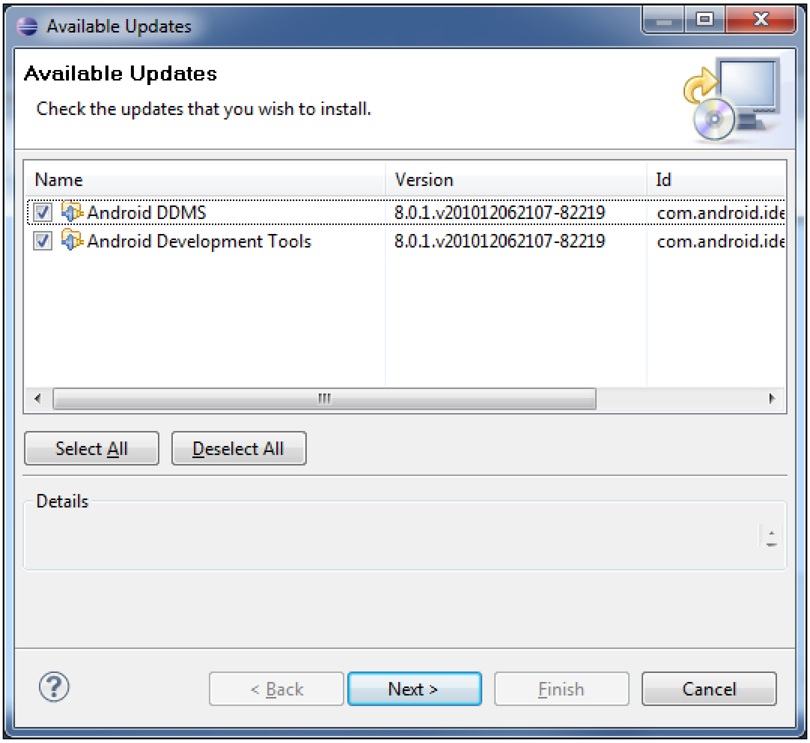


Abbildung 2.23: **Eclipse Update Manager**

Der Update-Manager kontaktiert die angegebenen Update Sites und zeigt verfügbare Aktualisierungen an. Per *Next* und *Finish* führen wir die Aktualisierung durch. Auch hier wird wieder vor *Unsigned Content* gewarnt, um die Aktualisierung zu installierenm bestätigen wir die Warnung mit *OK*. Nach der Aktualisierung sollte Eclipse neu gestartet werden.

Das Plug-in warnt nun, dass die neue Version der SDK-Tools noch nicht installiert ist. Das holen wir nun über den SDK-Manager nach.

Nach der Aktualisierung des ADT-Plug-ins erhalten wir einige neue Optionen im Android Repository. Bemerkenswert ist hier, dass auch Erweiterungen von Drittanbietern verfügbar sind, wie z.B. die Add-ons von Samsung. Samsung bietet hier Erweiterungen speziell für seine Tablet-Geräte an.

Der Einfachheit halber installieren wir einfach alle verfügbaren Packages, die noch nicht installiert und nicht als *(Obsolete)* gekennzeichnet sind. Dazu haken wir die Option *Display updates only* in der Schalterleiste des Dialogs an, wählen die Hauptpunkte aus und bestätigen mit *Install Selected*.

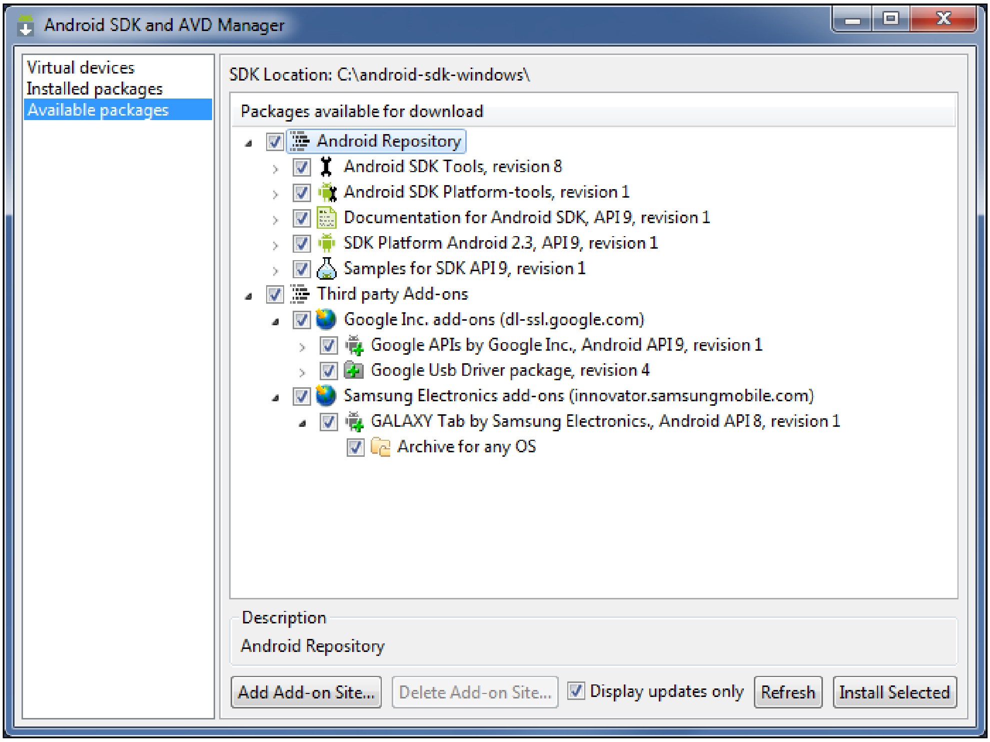


Abbildung 2.24: **Der Inhalt des Repository im SDK-Manager nach der Aktualisierung des ADT-Plug-ins**

Es folgt ein weiterer Dialog, in dem die Lizenzbedingungen der Packages akzeptiert werden müssen. Da wir alle Aktualisierungen installieren wollen, wählen wir die Option *Accept All* und bestätigen den Dialog mit *Install*.

Die Installation dauert eine Weile, aber nachdem wir uns in Geduld geübt haben stehen uns die neuen Funktionen zur Verfügung. Nach dem ersten Durchlauf, in dem grundlegende Aktualisierungen durchgeführt wurden, müssen wir noch einen Aktualisierungslauf starten, der letztendlich auch den SDK and AVD Manager aktualisiert. Danach sollte der SDK-Manager geschlossen werden.

ACHTUNG

Bei meiner Aktualisierung zeigt der SDK-Manager auch nach der Installation der Google- und Samsung-Add-ons diese weiterhin als verfügbare Updates an. Nachdem ich den Installationslauf wiederholt durchgeführt habe, muss ich zu dem Schluss kommen, dass der SDK-Manager hier nicht korrekt arbeitet.

Stellen wir uns aber schon jetzt mental darauf ein, diese Prozedur erneut durchzuziehen, wenn die Entwicklungslinien der 2.3 und 3.0 in der 2.4 oder 4.0 zusammengeführt werden ... oder so.

2.3 Android Development Tools im Detail

Die Android Development Tools bestehen neben den eigentlichen SDK-Komponenten aus weiteren Dokumenten und Werkzeugen, die für die Entwicklung wichtig und nützlich sind:

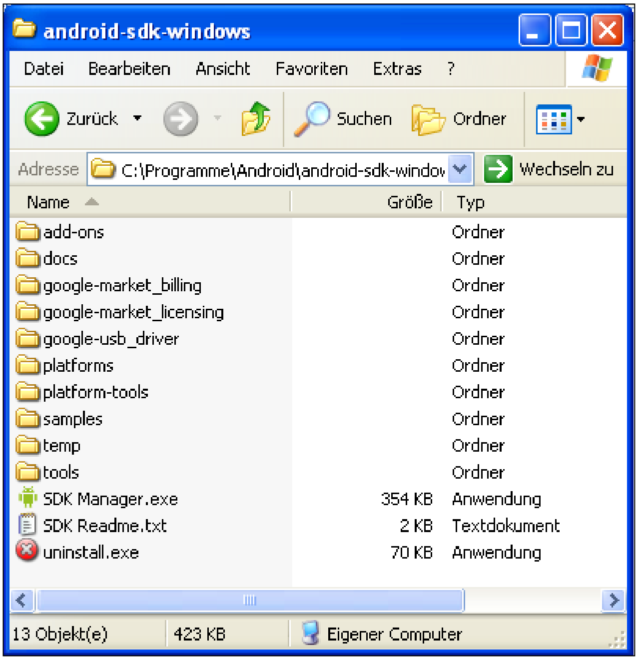


Abbildung 2.25: **Oberste Verzeichnisstruktur des Android-SDK**

Wichtig sind die Verzeichnisse `platform-tools` und `tools`. Hier befinden sich die Kommandozeilenwerkzeuge zum Kommunizieren mit der Android-Debug-Bridge und zum Verwalten virtueller Geräte sowie einige weitere nützliche Tools.

Da wir unsere Projekte in der Eclipse verwalten, haben wir relativ wenig mit den Werkzeugen direkt zu tun, da die meisten Tools über das Eclipse-Plug-in in der Entwicklungsumgebung eingebunden werden. Allerdings kann es manchmal notwendig sein, über die Werkzeuge direkt zu arbeiten, z.B. um Anwendungen vom Gerät zu entfernen oder die Android-Debug-Bridge neu zu starten.

add-ons	Hier befinden sich die Bibliotheken von Drittherstellern wie Google, Samsung, etc. **Innerhalb der Unterverzeichnisse** finden sich neben den Bibliotheken auch die Dokumentationen zu den Erweiterungen.
docs	SDK-Dokumentation. Entspricht der Doku unter `http://developer.android.com`. Sehr nützlich, wenn wir mal offline sind.

google-market_billing	Beinhaltet ein Beispiel für In-App-Billing (Abrechnung innerhalb einer Anwendung) sowie die Schnittstellendefinition, um mit dem Billing-Service des Android-Markets zu kommunizieren.
google-market_licensing	Bibliothek für die Nutzung des Android-Market-Lizenzierungsservice
google-usb-driver	Google-spezifische USB-Treiber für verschiedene Android Geräte (ADP1/T-Mobile G1, ADP2/Google Ion/T-Mobile myTouch 3G, Verizon Droid, Nexus One, Nexus S)
platforms	Emulator-Systemimages und Android-Bibliotheken für die unterschiedlichen API-Levels
platform-tools	Werkzeuge für diese Entwicklungsplattform. Beinhaltet unter anderem `adb.exe` und die Packaging-Tools zum Erzeugen der `.apk`-Files (Android Package File) sowie den Dalvik-Cross-Assembler `dx.exe`.
samples	Beispielprojekte für die verschiedenen API-Levels. Diese Beispielprojekte lassen sich mittels des Eclipse ADT-Plugins in den Eclipse Workspace importieren.
temp	Temporäres Verzeichnis.
tools	Verschiedene weitere Werkzeuge. Beinhaltet die Werkzeuge, um die Android-Projekte auf der Kommandozeile zu verwalten, und weitere Hilfsmittel.

2.3.1 Der SDK- und AVD-Manager

Wie wir bereits festgestellt haben, ist der SDK and AVD Manager eins der ersten Tools mit dem wir es nach der Installation zu tun haben. Hiermit werden neue Komponenten installiert und bestehende aktualisiert und die virtuellen Geräte (*Android Virtual Device – AVD*) verwaltet.

Der SDK and AVD Manager wird entweder über das Windows-Startmenü (*Start → Alle Programme → Android-SDK Tools → SDK Manager*) oder innerhalb der Eclipse über das Android-Icon oder über *Window → Android-SDK and AVD Manager* aufgerufen.

Die Installation von Komponenten mit dem SDK and AVD Manager haben wir bereits kennengelernt.

Die weitere essenzielle Funktion des SDK and AVD Managers ist das Erstellen und Verwalten der virtuellen Geräte, der virtuellen Devices.

Virtuelle Geräte können in jeder denkbaren Ausstattung erstellt und über einen Emulator ausgeführt werden. Der Emulator ist ein vollwertiges Android-System und wird beim Erstellen des Geräts auch mit dem entsprechenden Android-Image erzeugt (Version 1.5, 2.1, 3 …).

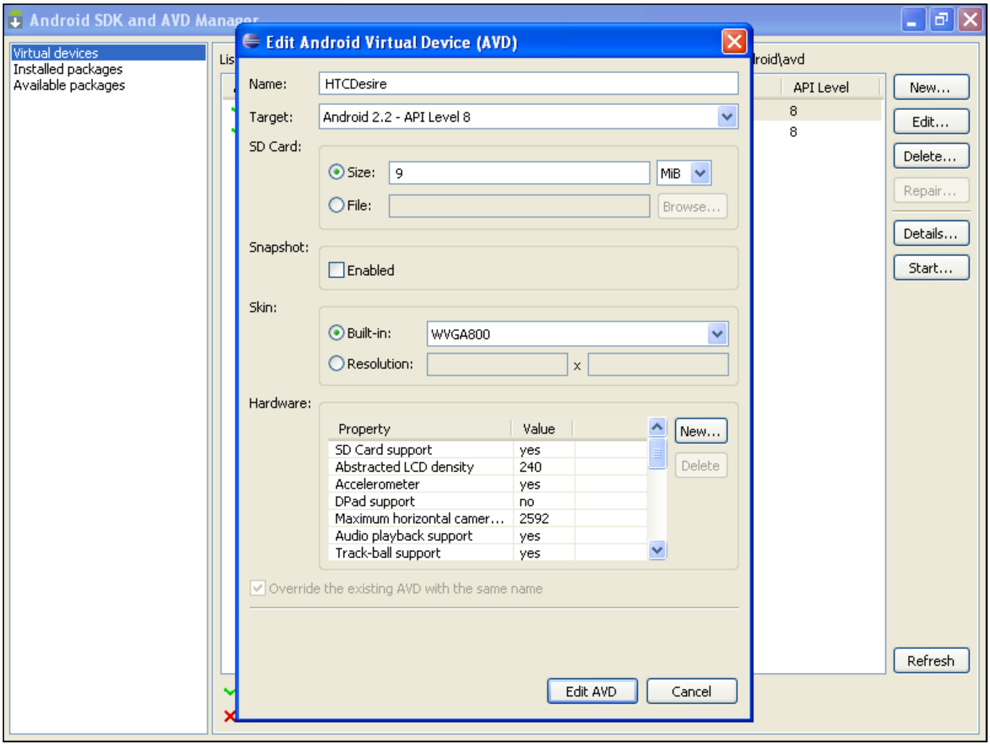


Abbildung 2.26: **Erstellen eines virtuellen Android-Geräts**

INFO

Das ist so ziemlich die erste Begegnung mit dem API-Level. Im Feld *Target* tragen wir ein, unter welcher Android-Version das Gerät laufen soll. Später, wenn wir Projekte erstellen, müssen wir den API-Level für das Projekt angeben. Damit kann das Projekt dann nur auf Geräten mit gleichem oder höherem API-Level laufen, und wir erhalten später bei der Geräteauswahl auch nur die Geräte angezeigt, die dem API-Level unserer Anwendung entsprechen.

Mittels der Android-Debug-Bridge können Ereignisse im virtuellen Gerät erzeugt werden, die normalerweise »von außen« kommen, wie SMS, eingehende Telefonanrufe oder GPS-Datenströme.

Die Eckdaten der virtuellen Geräte kann man über die Herstellerwebseiten ermitteln oder auch über Wikipedia, wenn es zu dem jeweiligen Gerät einen Artikel gibt.

TIPP

Ich habe ein paar AVDs auf die beiliegende CD gepackt. Die Dateien liegen unter `Virtual Devices\avd` und können einfach in das entsprechende Verzeichnis kopiert werden. Die »naturgetreue« Abbildung des jeweiligen realen Geräts ist nicht in allen Fällen gegeben.

Die virtuellen Geräte werden im jeweiligen Benutzerverzeichnis unter `C:\Dokumente und Einstellungen\<Benutzer>\.android\avd` (Windows XP) bzw. `C:\Users\<Benutzer>\.android\avd` (Windows Vista/7) gespeichert.

INFO

Für jedes virtuelle Gerät wird dort in einem Unterverzeichnis ein Image erzeugt, das den internen Speicher für User-Daten repräsentiert sowie, falls eine SD-Card angegeben wird, ein entsprechendes SD-Card-Image mit der jeweilgen SD-Kartengröße.

Die Betriebssystem-Images selbst liegen unter `C:\Programme\Android\android-sdk-windows\platforms\android-<API Level>\images`.

Das virtuelle Gerät nimmt dann schon mal einen gewissen Platz auf der Festplatte ein.

Die Eckdaten des Geräts werden in einer `.ini`-Datei abgelegt die auch immer im Standardverzeichnis verbleibt:

```
C:\Dokumente und Einstellungen\<Benutzer>\.android\HTCDesire.ini
```

TIPP

Um die Daten-Images in einem anderen Verzeichnis zu speichern muss man auf der Kommandozeile im Verzeichnis `C:\Programme\Android\android-sdk-windows\platforms` **mit dem Werkzeug** `android` **(**`android.bat`**) arbeiten:**

```
android move avd -n HTCDesire -p d:\AVD_Images
```

Die Einstellungsdateien (`HTCDesire.ini`**) verbleiben allerdings im …**`\.android\avd`**-Verzeichnis.**

Starten lassen sich die Geräte entweder aus dem SDK and AVD Manager oder aber aus der Eclipse heraus über das Menü *Run* → *Debug Configurations*.

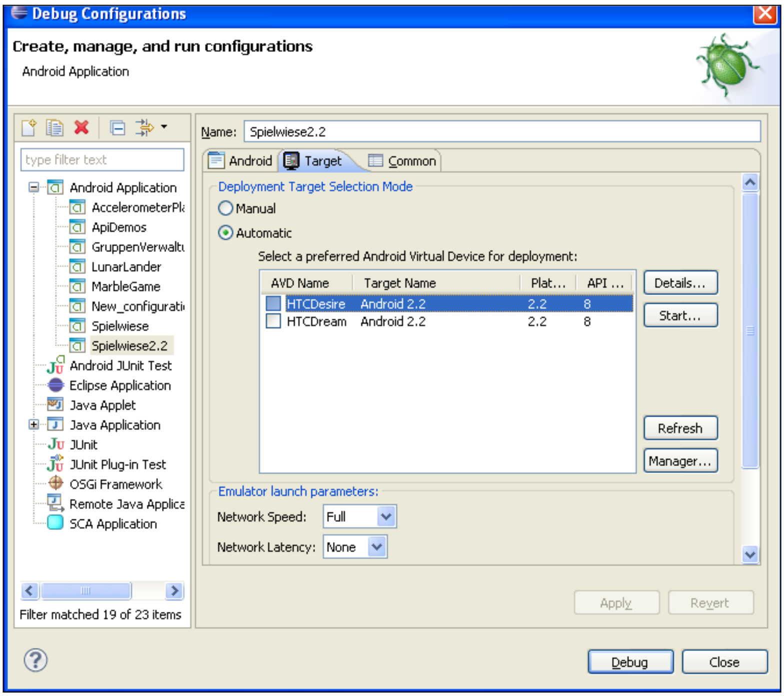


Abbildung 2.27: **Auswahl des virtuellen Geräts zum Testen einer Anwendung**

ACHTUNG

Manchmal sind die Knöpfe in diesem Dialog alle ausgegraut. In dem Fall einfach mal auf *Manual* und dann wieder auf *Automatic* klicken, dann erhält man Zugriff auf die Knöpfe.

Über die Option *Automatic* können wir dem Projekt das bevorzugte virtuelle Gerät zuweisen und später immer mit diesem starten. Wenn wir allerdings gerne die Wahl haben möchten, benutzen wir die Option *Manual*. Betätigen wir mit der Option *Manual* den Knopf *Debug*, können wir aus den (für den API-Level) verfügbaren virtuellen **und angeschlossenen echten** Geräten auswählen.

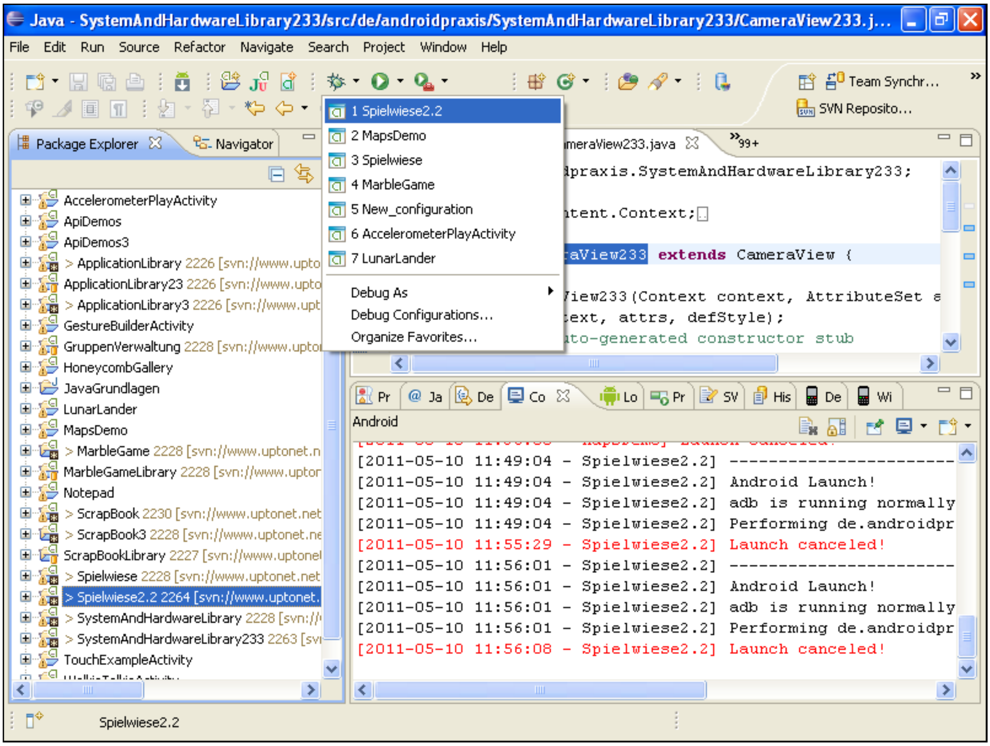


Abbildung 2.28: **Auswahl des Projekts »Spielwiese2.2« zum Testen**

Hier sehen wir die für unterschiedliche Projekte erstellten Debug-Konfigurationen. Das ausgewählte Projekt »Spielwiese2.2« adressiert den API-Level 8 (Android 2.2).

Wählen wir dieses Projekt zum Testen, erhalten wir die mit diesem API-Level kompatiblen Geräte zur Auswahl.

Hier sehen wir, dass alle erstellten AVDs das Projekt ausführen können und dass ein echtes Gerät per USB angeschlossen ist, auf dem wir das Projekt ebenfalls ausführen könnten.

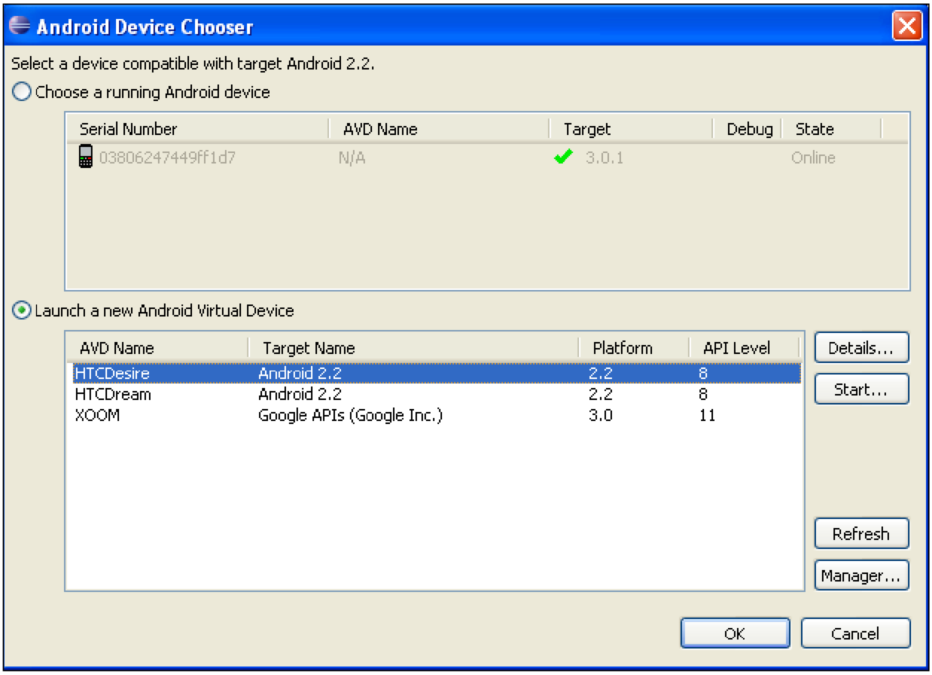


Abbildung 2.29: **Auswahl eines kompatiblen Geräts**

Hier starten wir das virtuelle Gerät, das in etwa einem HTC-Desire entspricht. Der Startvorgang kann je nach virtuellem Gerät einige Minuten dauern, in der Eclipse können wir beobachten, wie das Gerät gestartet, die Anwendung installiert und zur Ausführung gebracht wird:

Listing 2.1: **Ausgabe der Eclipse beim Starten der Anwendung**

```
[2011-05-10 11:56:38 - Spielwiese2.2] Android Launch!
[2011-05-10 11:56:38 - Spielwiese2.2] adb is running normally.
[2011-05-10 11:56:38 - Spielwiese2.2] Performing de.androidpraxis.Spielwie➥
    se.Spielwiese activity launch
[2011-05-10 12:02:36 - Spielwiese2.2] Launching a new emulator with Virtual
Device ‚HTCDesire'
[2011-05-10 12:03:23 - Spielwiese2.2] New emulator found: emulator-5554
[2011-05-10 12:03:23 - Spielwiese2.2] Waiting for HOME (‚android.process.
acore') to be launched...
[2011-05-10 12:06:05 - Spielwiese2.2] HOME is up on device ‚emulator-5554'
[2011-05-10 12:06:05 - Spielwiese2.2] Uploading Spielwiese2.2.apk onto de➥
    vice ‚emulator-5554'
[2011-05-10 12:06:05 - Spielwiese2.2] Installing Spielwiese2.2.apk...
[2011-05-10 12:06:58 - Spielwiese2.2] Success!
[2011-05-10 12:07:00 - SystemAndHardwareLibrary] Could not find SystemAnd➥
    HardwareLibrary.apk!
[2011-05-10 12:07:00 - Spielwiese2.2] Starting activity de.androidpraxis.
Spielwiese.Spielwiese on device emulator-5554
[2011-05-10 12:07:28 - Spielwiese2.2] Attempting to connect debugger to ‚de.
androidpraxis.Spielwiese' on port 8616
```

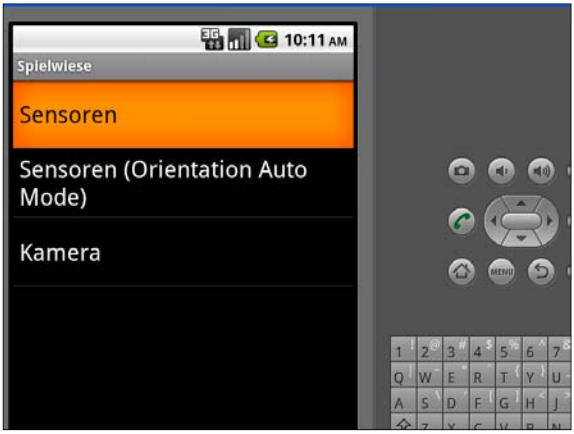


Abbildung 2.30: **Gestartetes virtuelles Gerät**

2.3.2 Anschluss von Android-Geräten über USB

Am schönsten ist es, wenn man ein »richtiges« Android-Gerät zur Verfügung hat, mit dem man seine Anwendungen testen kann. Die Geräte werden mittels USB über die *Android-Debug-Bridge* (*adb*) an den Host – das ist unser Entwicklungscomputer – angeschlossen.

Damit die *adb* die Geräte auch richtig erkennt ist in den meisten Fällen noch ein spezieller USB-Treiber zu installieren, da standardmäßig nur die USB-Treiber zum Anschluss des Geräts als Massenspeicher, also zum Zugriff auf das Dateisystem, installiert werden.

Hat man diesen speziellen USB-Treiber **nicht** installiert, findet sich das Gerät nicht in der Liste der laufenden Geräte des Device-Managers, und man kann folglich auch seine Anwendung nicht hochladen und testen.

Für einige Smartphones wie das T-Mobile G1/G3, Verizon Driod, Nexus One und Nexus S oder ähnliche Geräte anderer Anbieter liefert das SDK die USB-Treiber mit.

Unter `http://developer.android.com/sdk/win-usb.html` finden wir die entsprechenden Informationen dazu und einen nützlichen Link auf eine Liste von Bezugsquellen für OEM USB Treiber anderer Hersteller wie Acer, Motorola, Dell, HTC und andere: `http://developer.android.com/sdk/oem-usb.html`.

Wenn wir Geräte verwenden, die mit den Google-Treibern auskommen, reicht es in der Regel aus, das Gerät anzuschließen. Der Treiber wird durch die Hardwareerkennung automatisch installiert.

Sollte das mal nicht funktionieren und der Treiber manuell gesucht werden müssen: Die Google-Treiber befinden sich unter `C:\Programme\Android\android-sdk-windows\google-usb_driver`.

Bei allen anderen Herstellern lädt man in der Regel eine Setup-Datei herunter, die die Treiber installiert.

TIPP

Wir sollten beim Installieren des Treibers das Gerät nicht angeschlossen haben. Die Setup-Programme kopieren die Treiber in die dafür vorgesehenen Windows-Verzeichnisse und machen sie damit für Hardwareerkennung verfügbar. Die springt aber erst an, wenn wir das Gerät erneut anschließen.

2.3.3 9-Patch-Zeichenprogramm

Unter 9-Pach-Bildern (Drawables) versteht man PNG-Bitmaps, die in vertikaler und horizontaler Richtung gestreckt werden können, ohne dass sie »pixelig« werden. Die Bitmaps müssen so gestaltet sein, dass sie sich in neun Felder (deshalb 9-Patch) zerlegen lassen. Die vier entstehenden Ecken werden dann beim Vergrößern im entstehenden Rechteck wieder in die neuen Ecken gesetzt, die restlichen Felder entsprechend in vertikaler Richtung und horizontaler Richtung dupliziert. Dadurch findet keine Vergrößerung des Bildes statt, sondern die »ähnlichen« Teile werden einfach dupliziert, und die Skalierung erscheint verlustfrei.

Diese Technik ist besonders gut für Buttons und flexible Hintergründe geeignet, die sich an den Inhalt, Schriftgröße oder die Bildschirmgröße dynamisch anpassen sollen.

Mit dem 9-Patch-Zeichenprogramm können für entsprechend gestaltete PNG-Bitmaps die Patch-Felder eingezeichnet werden.

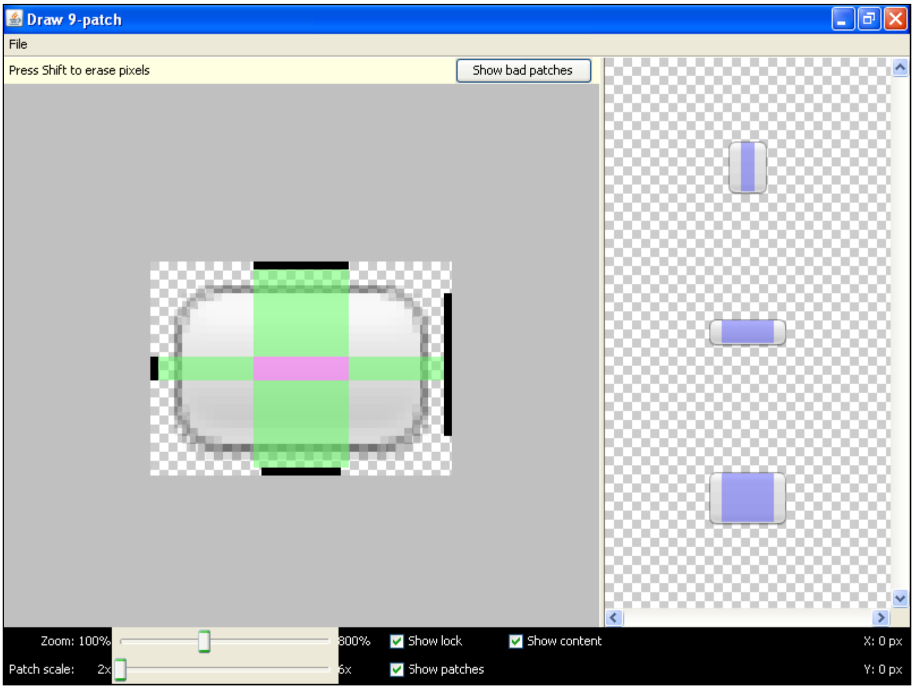


Abbildung 2.31: **Draw 9-Patch**

Das Werkzeug findet sich im Verzeichnis `C:\Programme\Android\android-sdk-windows\tools`.

2.3.4 Android Debug Bridge

Die Android Debug Bridge `adb.exe` dient als »Brücke« zwischen dem Entwicklungscomputer (Host) und dem virtuellen oder realen Android-Gerät. Über die Debug Bridge lassen sich verschiedene Kommandos an die Geräte senden, Daten zwischen dem Gerät und dem Host austauschen und vieles mehr.

In der Regel nutzen wir die Debug Bridge über das ADT-Plug-in in Verbindung mit dem DDMS (Dalvik Debug Monitor Server) direkt in Eclipse, es gibt aber auch Situationen, in denen die direkte Nutzung auf der Kommandozeile mehr Optionen bietet. Log-Dateien lassen sich z.B. direkt im Logcat-Window des Plug-ins erstellen, per `adb.exe` (`adb logcat <optionen>`) haben wir aber mehr Kontrolle über das Schreiben der Log-Files.

Die detaillierte Erklärung der einzelnen adb-Kommandos würde den Rahmen dieses Buches sprengen, wenn wir wirklich so in die Tiefen einsteigen wollen, führt erst einmal an der Originaldokumentation kein Weg vorbei: `http://developer.android.com/guide/developing/tools/adb.html`.

Hier finden wir Informationen zu Logcat:

`http://developer.android.com/guide/developing/tools/logcat.html`.

und

`http://developer.android.com/guide/developing/debugging/debugging-log.html`.

In diesem Zusammenhang ist es interessant zu wissen, dass ein Android-Gerät Ringpuffer für Telefonie- und andere Ereignisse anlegt, die mit `adb logcat -b radio` bzw. `adb logcat -b events` ausgelesen werden können.

Das Werkzeug findet sich im Verzeichnis `C:\Programme\Android\android-sdk-windows\platform-tools`.

2.3.5 Das ADT-Plug-in

Das ADT-Plug-in für Eclipse vereint die Entwicklungswerkzeuge und weitere Hilfsmittel zum Erstellen und Testen von Android-Anwendungen innerhalb der Eclipse. Nur in Ausnahmefällen müssen wir auf die Werkzeuge über die Kommandozeile zugreifen.

Eclipse arbeitet mit sogenannten Perspektiven und Sichten (Perspective and View). Eine Perspektive wird aus Views aufgebaut, die thematisch zusammenhängen.

Die Perspektiven können über die Perspektiven-Umschaltung rechts oben bzw. über *Window* → *Open Perspective* → *Other*...gewechselt werden.

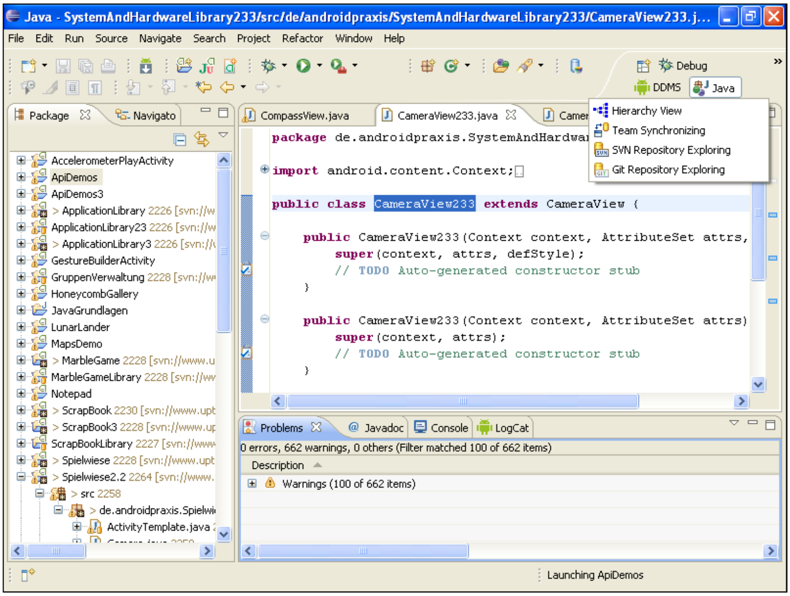


Abbildung 2.32: **Wechseln der Perspektive**

Java View

In der *Java View* erstellen wir unsere Programme. Hier legen wir neue Projekte an, bearbeiten die Quelltextdateien und bearbeiten auch unter Zuhilfenahme des ADT-Plug-ins die Android-Ressourcendateien mittels spezieller Editoren.

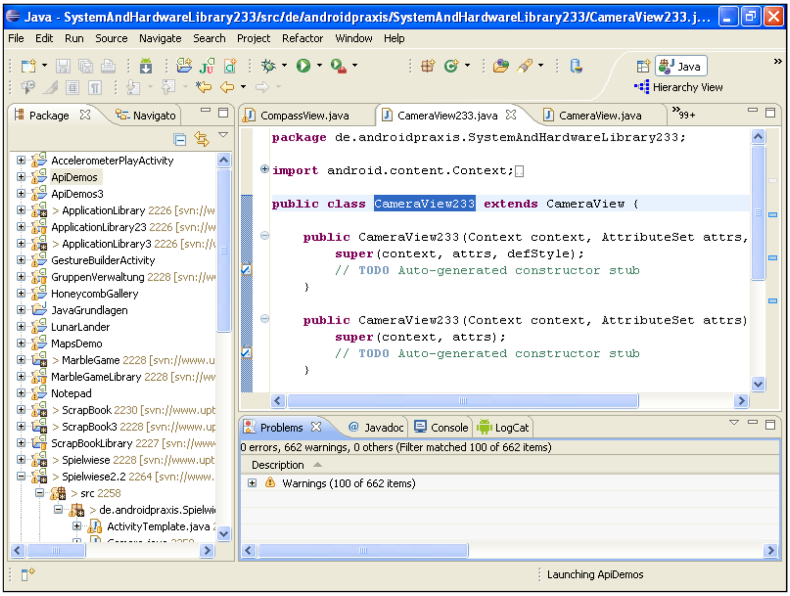


Abbildung 2.33: **Die Java-Perspektive**

Debug View

In der Debug View testen wir unsere Programme. Wenn ein Programm in einem virtuellen Gerät oder einem echten Gerät über die Eclipse gestartet wurde, können wir in der Debug View Haltepunkte setzen, um die Ausführung an einer bestimmten Stelle zu unterbrechen, Variableninhalte ansehen und die Log-Ausgaben im Logcat-Fenster anschauen.

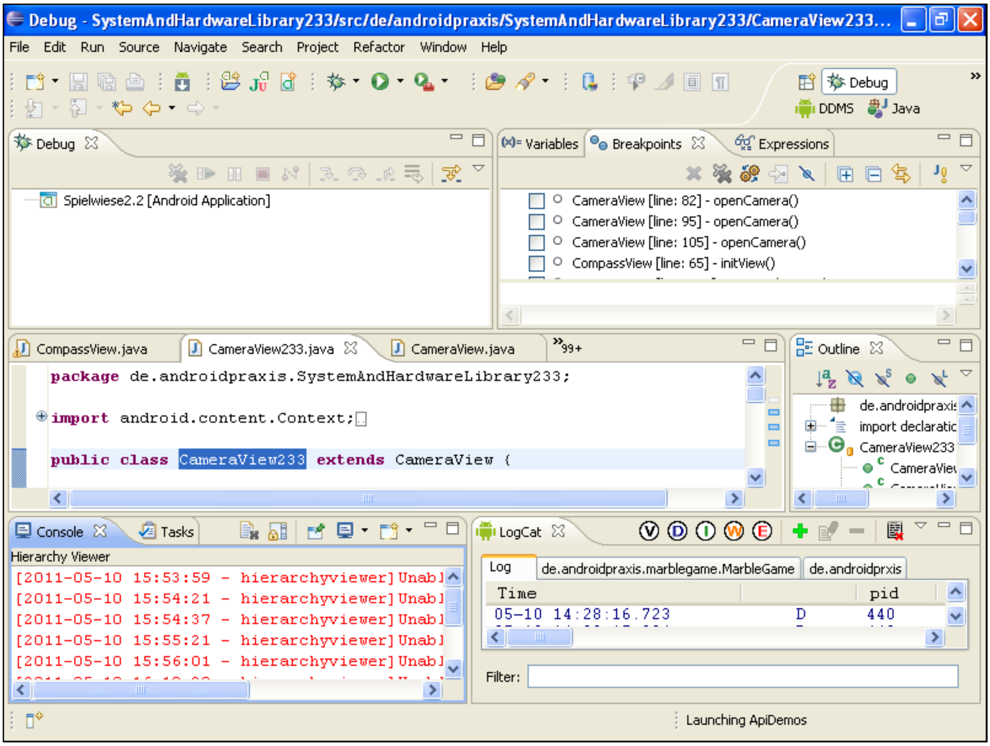


Abbildung 2.34: **Die Debug-Perspektive**

Die DDMS-Perspektive

Die DDMS-Perspektive (Dalvik Debug Monitor Server) bietet uns den Zugang zum Inneren der Android-Geräte, egal ob es sich um ein virtuelles oder reales Gerät handelt.

Hier haben wir einen Überblick über die laufenden Geräte, können uns das Dateisystem anschauen, die Speicherbelegung nachverfolgen und noch vieles mehr.

Über die DDMS-Perspektive erhalten wir Zugriff auf das komplette interne Dateisystem. Im Gegensatz dazu erhalten wir beim Anschluss des Geräts als portables Device bzw. Massenspeicher lediglich den Zugriff auf den Teilbaum `/sdcard` des Dateisystems.

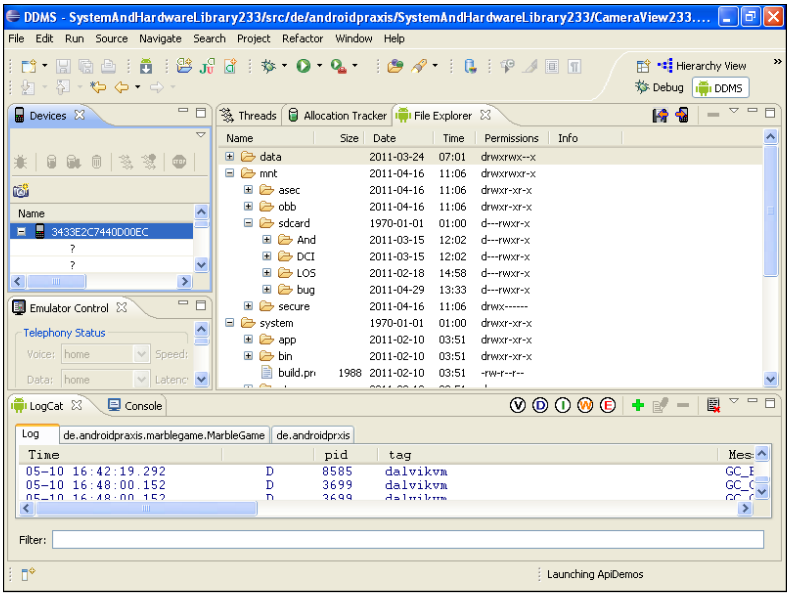


Abbildung 2.35: **Die DDMS Perspektive**

Besonders interessant ist hier die Möglichkeit mit dem Kamerasymbol Bildschirmfotos der laufenden Geräte anzufertigen.

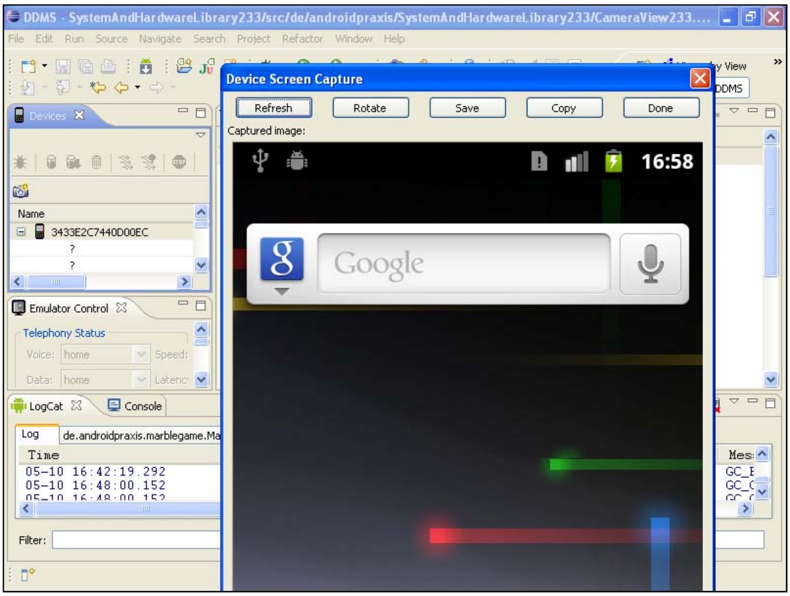


Abbildung 2.36: **Screen-Capture (Bildschirmfoto) des Google Nexus S**

Die Hierarchy View

Hier kann man die »Fenster« der Benutzeroberfläche untersuchen und die Views in einem hierarchischen Viewer anzeigen lassen, ebenfalls Screenshots anfertigen und die Oberfläche im Detail mit einer Lupe untersuchen.

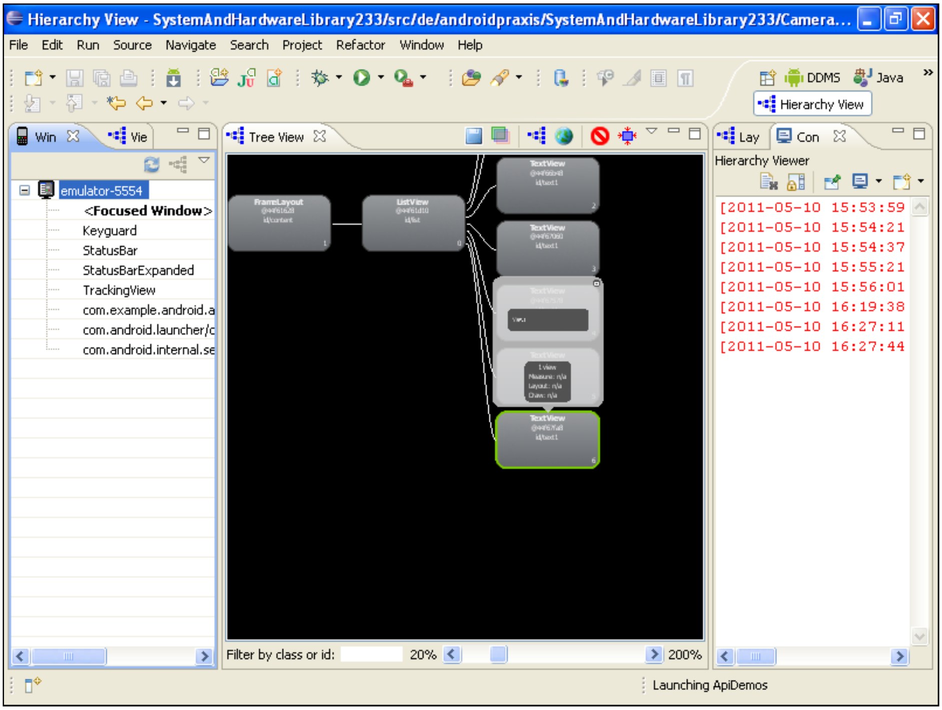


Abbildung 2.37: **Die Hierarchy View**

ACHTUNG

Aus Sicherheitsgründen geht das anscheinend nicht bei allen Geräten, sondern nur auf Geräten mit einem speziellen Entwickler-Kernel. Ich habe den Hierarchie-Viewer nur auf den Emulatoren zum Laufen bekommen. Bei den echten Geräten, auf denen ich das ausprobiert habe, kam die Fehlermeldung `[2011-04-10 17:09:45 - hierarchyviewer]-Unable to debug device XXXXXXXXXXXX`

2.4 Fazit

Wir haben nun das Android-SDK, die Werkzeuge und die Entwicklungsumgebung installiert und ein wenig kennen gelernt. Wie wir gesehen haben, liefern die Developer Tools eine Menge Werkzeuge mit, sowohl für die Kommandozeile als auch hübsch verpackt im Eclipse-Plug-in. Es gäbe schon allein zu den einzelnen Werkzeugen eine Menge zu sagen, und einiges werden wir auch während der Programmierung noch kennen lernen. Wie bei allen neuen Dingen gilt auch hier der Spruch: Übung macht den Meister.

Und nun können wir endlich mit der Programmierung anfangen.

3 Android – Schritt für Schritt



Wir haben es geschafft. Die Entwicklungsumgebung läuft, und wir können uns mit unserer eigentlichen Aufgabe beschäftigen: die Programmierung unter Android zu erforschen und kennenzulernen.

In diesem Kapitel beschäftigen wir uns mit dem grundlegenden Aufbau von Android-Applikationen und lernen gleichzeitig, wie Projekte angelegt werden, aus welchen Komponenten ein Android-Projekt besteht, wie man ein Programm zum Laufen bringt und testet. Darüber hinaus werden wir uns auch mit den wichtigen Grundlagen zur Signatur der Programme und den Richtlinien (Policies), denen Programme genügen sollen, sowie dem Manifest beschäftigen.

3.1 Anlegen eines Projekts

Ein Android-Projekt ist in Eclipse schnell angelegt. Aus dem Hauptmenü wählen wir *File* → *New* → *Android Project*. Alternativ lässt sich das über die Werkzeugleiste erledigen. Sollte die Option *Android Project* nicht vorhanden sein, ggf. weil die Eclipse über mehrere Projekttypen verfügt, gelangen wir über *File* → *New* → *Project...* zu einer Übersicht über alle in Eclipse enthaltenen Projekttypen.

Im folgenden Dialog müssen wir einige Eckdaten zu unserem Projekt erfassen. Neben dem Namen gehören dazu z.B., für welche Plattform wir die Applikation entwickeln, und die Angabe darüber, welche SDK-Version wir mindestens für unser Projekt voraussetzen.

3.1.1 Das Projekt

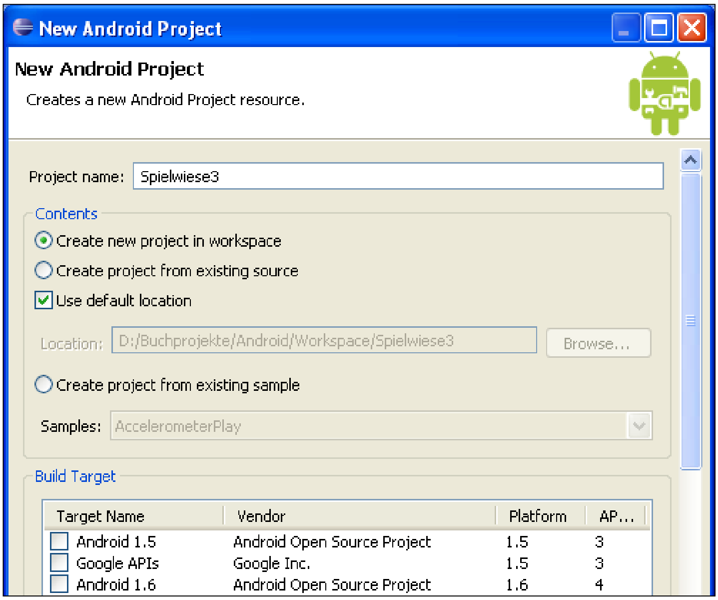


Abbildung 3.1: **Projekt anlegen**

Unter *Project Name* geben wir den Namen des Projekts an. Da unter diesem Namen das Projekt als Verzeichnis in unserem Workspace gespeichert wird, muss dieser Name einem gültigen Verzeichnisnamen entsprechen. Ich nenne das Projekt hier *Spielwiese3*, die Spielwiese wird uns durch dieses Kapitel begleiten.

3.1.2 Build Target

Das *Build Target* bestimmt, für welche Android-Version wir entwickeln wollen. Je nach dem, was wir hier auswählen, haben wir die Funktionen der jeweiligen Betriebssystemversion zur Verfügung.

Die letzte Spalte der Build Targets zeigt den API-Level. Bei allen Dingen, die sich auf die Betriebssystemversion beziehen, wird immer der API-Level angegeben und niemals die *Platform*.

ACHTUNG

Die Auswahl des Build-Targets bestimmt, welche Funktionen des Betriebssystems unsere Applikation erwartet. Allerdings kann eine Applikation für den API-Level 7 durchaus auf einem Gerät mit dem API-Level 3 installiert werden, wird aber abstürzen, sobald Funktionen aus einem höheren Level angesprochen werden.

Wichtig wird in diesem Zusammenhang der Parameter *Min SDK Version*. Mit diesem Parameter teilen wir dem *Packaging-System* mit, welche SDK-Version wir mindestens voraussetzen. Android verhindert dann die Installation der Applikation auf einem niedrigeren API-Level.

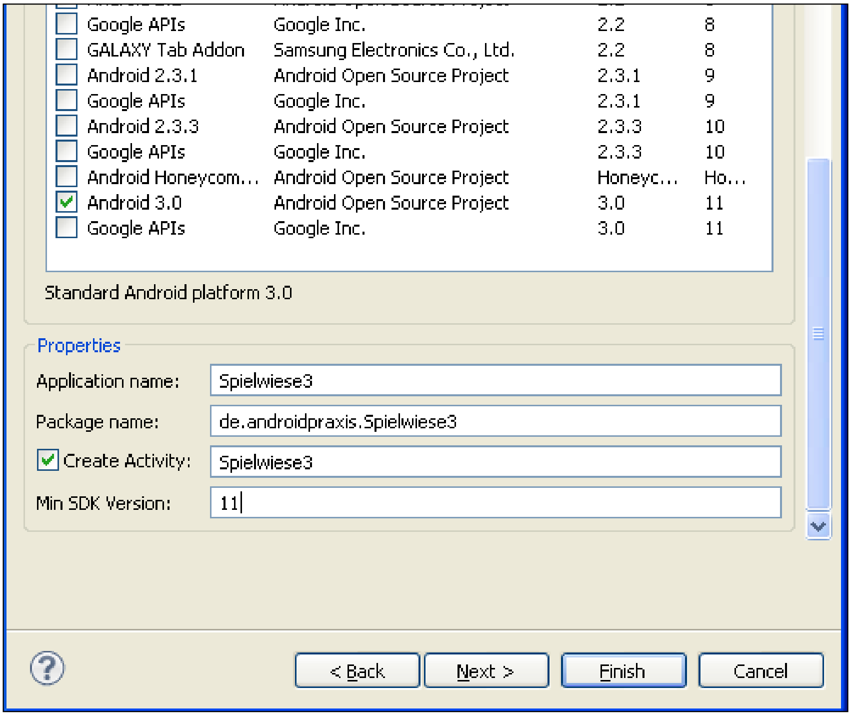


Abbildung 3.2: **Projekt anlegen – weitere Optionen**

Für die Spielwiese wählen wir Android 3.0 als *Build Target*. In der Abbildung sehen wir, dass es für den API-Level 11 zwei Optionen gibt. Wie wir bereits diskutiert haben, ist Android ein offenes System, das durch Gerätehersteller und Softwareanbieter erweitert werden kann. Diese Erweiterungen sind dann oft Bestandteil des erweiterten Frameworks oder zusätzliche Bibliotheken. Die Erweiterungen können über die Update-Sites der Hersteller mittels des *SDK and AVD Managers* zur Verfügung gestellt werden und tauchen dann ebenfalls hier in der Liste der *Build Targets* auf. Hier sehen wir, dass Google die *Google APIs* bereitstellt, und zwar komplett als *Build Target* und nicht lediglich als zusätzliche Bibliothek. Das heißt, hier hat Google das Framework um eigene Komponenten erweitert, die aber ebenfalls auf Geräten mit dem API-Level 11 laufen. Die *Google APIs* beinhalten Erweiterungen zum Zugriff auf Google Maps.

3.1.3 Application und Package Name

Der *Application Name* ist der Name unserer Applikation und wird als Vorgabe für alle *Application Components* (Activities, Services, BroadcastReceiver und ContentProvider) im Manifest gesetzt. Dieser Name wird z.B. in der Titelleiste einer Activity angezeigt.

Die einzelnen Komponenten können im Manifest auch eigene Namen erhalten.

Der Name der Anwendung ist auch der Name, der bei Verwaltung installierter Anwendungen angezeigt wird, und muss nicht unbedingt mit dem Namen übereinstimmen, der auf einer *Home-Screen* oder im *Application Launcher* angezeigt wird. Meistens wird es aber so sein, dass zumindest die Haupt-Activity der Anwendung genauso heißt wie die Anwendung.

Der Name wird bereits als String-Ressource abgespeichert und kann damit leicht an andere Sprachen (*Internationalisiert*) angepasst werden. Was es damit genau auf sich hat, werden wir im Laufe dieses Kapitels noch sehen.

Der *Package Name* bestimmt zum einen den Namensraum unserer Applikation, zum anderen auch die Organisation unserer Java-Klassen im Projekt. Der *Package Name* wird zum *Root-Package* unseres Projekts, alle weiteren Packages und Klassen werden unter diesem *Root-Package* angelegt. Im Manifest wird das Package ebenfalls aufgeführt, und alle unsere Application Components verhalten sich relativ zu diesem Package.

Die Packages werden wie ein umgekehrter Domain-Name gebildet.

INFO

In Java werden die Klassen und Schnittstellen in Paketen abgelegt. Die Pakete spannen zum einen den Verzeichnisbaum des Java-Projekts auf, zum anderen sind die Pakete auch die Namensräume in denen sich die Klassen und Schnittstellen tummeln. Eine Klasse hat einen Klassennamen und liegt in einem Package. Der vollqualifizierte Name (Full qualified Name) der Klasse ergibt sich aus dem Package in Verbindung mit dem Klassennamen.

Der *Package Name* sollte eindeutig sein. In der Regel werden Packages nach den Domain-Namen der Hersteller bzw. Autoren der Programme organisiert:

Hersteller: Google

Domain-Name: google.com

Package: com.google.app.*application_name*

Autor: Mike Bach

Domain-Name: androidpraxis.de

Package: de.androidpraxis.Spielwiese3

3.1.4 Create Activity

Wenn wir diese Option anhaken, dann erstellt der Projekt-Assistent eine *Activity* unter dem angegebenen Namen (hier: Spielwiese3). Eine *Activity* ist die Anwendungskomponente, die dem Anwender die Oberfläche der Anwendung zeigt und mit der der Anwender interagieren kann. Genaueres dazu und zu anderen Anwendungskomponenten erfahren wir im Laufe dieses Kapitels.

Der Projekt-Assistent erstellt eine Klasse mit dem gegebenen Namen innerhalb des Packages, das wir für unsere Anwendung benannt haben.

3.1.5 Min SDK Version

Neben der Plattform ist auch der APL-Level wichtig. Der API-Level bestimmt die aktuelle Version des Android-Frameworks. Das Framework ist eng verzahnt mit dem zugrundeliegenden Android-System und bildet die Schnittstelle zwischen der Applikation und dem eigentlichen Betriebssystem. Die APIs der unterschiedlichen Level sind immer abwärtskompatibel, aber nicht aufwärtskompatibel. Nutzen wir also spezielle Funktionen eines höheren API-Levels, führt die Verwendung dieser Funktionen auf einem niedrigeren API-Level zum Fehler.

Android verhindert jedoch nicht automatisch die Installation einer Anwendung, die mit einem höheren API-Level erstellt wurde. Daher ist es wichtig, die *Min SDK Version* bei der Projektanlage anzugeben. Damit erreichen wir zweierlei:

1. Die Applikation kann auf Geräten mit niedrigerem API-Level nicht installiert, somit auch nicht ausgeführt werden und keinen Absturz verursachen.
2. Im Android-Market wird die Applikation überhaupt nur dann angezeigt, wenn das Gerät mindestens über den API-Level verfügt. Damit wird der Anwender aktiv davor geschützt, Applikationen zu laden, die für sein Gerät nicht geeignet sind.

ACHTUNG

In das Feld Min SDK Version gehört tatsächlich der ganzzahlige Wert des API-Levels, nicht (wie der Name vermuten ließe) die Versionsnummer der Plattform.

Wenn wir als *Min SDK Version* einen API-Level wählen, der kleiner ist als der API-Level der gewählten Plattform, erhalten wir im Dialog einen Warnhinweis – ziemlich versteckt ganz oben im Titelbereich.

Auf der sicheren Seite sind wir immer dann, wenn wir speziell für ein Target und genau diesen API-Level entwickeln. Dann können wir davon ausgehen, dass auf neueren Versionen die Applikation immer noch läuft, wir müssen uns aber keine Gedanken darum machen, dass wir Funktionen höherer Versionen verwenden die ggf. auf Geräten mit kleinerem API-Level nicht laufen. Das ist alles eine Frage der Mühe und Sorgfalt, die wir investieren wollen. Ich möchte hier Techniken entwickeln, wie wir Applikationen so organisieren können, dass wir mehrere API-Levels abdecken können. Denn gerade zu Zeiten, wo eine neue Android-Version veröffentlicht wird, die neue spannende Funktionen enthält, dürfen wir die vielen Geräte, die eine niedrigere Version nutzen nicht vernachlässigen.

3.1.6 Erstellen des Projekts

Nachdem alle Angaben gemacht sind, können wir entweder mit *Next* fortfahren oder mit *Finish* das Projekt erstellen lassen.

Mit *Next* können wir im nächsten Schritt ein sogenanntes Testprojekt erstellen lassen. In einem Testprojekt können Testfälle für automatische Tests definiert werden.

Nach dem Betätigen von *Finish* wird das Projekt erzeugt und im Package Explorer von Eclipse dargestellt.

3.2 Die Projektstruktur

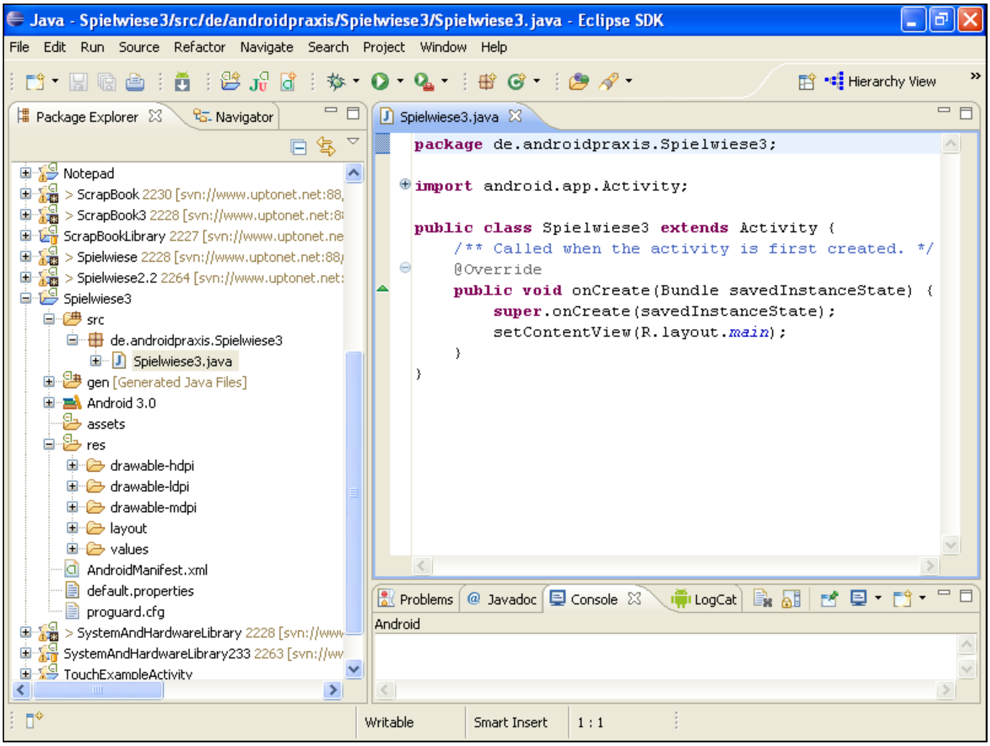


Abbildung 3.3: **Die Projektstruktur**

Im Package Explorer erkennen wir sehr schön die Struktur unseres Spielwiesen-Projekts, nämlich den Projektnamen *Spielwiese3*, das Package `de.androidpraxis.Spielwiese3` im Ordner `src` und die Klasse `Spielwiese3.java`.

Diese Angaben stimmen mit dem *Project name*, dem *Package name* und der *Create Activity*-Option überein.

Der Ordner `src` ist der Quellordner, in dem wir alle Java-Quellen unseres Projekts organisieren, und zwar innerhalb des Pakets `de.androidpraxis.Spielwiese3`.

Im Ordner `gen` legt das *Android Asset Packaging Tool* (*aapt*) die Ressourcen-IDs innerhalb der Klasse `R.java` ab, die ebenfalls in unserem Package platziert wird. In diesem Verzeichnis werden wir selbst nichts anlegen und auch die dort hinterlegten Java-Dateien nicht per Hand verändern, da die Klassen immer wieder neu erzeugt werden.

Der Ordner `Android 3.0` beinhaltet die Android-Bibliothek, die wir verwenden. Wir haben als *Build Target* den API-Level 11 gewählt, was der Android-Version 3.0 entspricht, womit die entsprechende Bibliothek in unser Projekt aufgenommen wird.

Java-Bibliotheken sind JAR-Dateien, die ähnlich einem ZIP-Archiv die Verzeichnis- bzw. Package-Struktur eines Java-Projekts enthält, und zwar sowohl die kompilierten Klassen als auch benötigte Ressourcen, manchmal sogar den zugehörigen Java-Quellcode.

INFO

Wir können auf Klassen aus solchen Bibliotheken zugreifen, indem wir das entsprechende Package importieren (z.B. `import android.app`**). Unsere Activity** `Spielwiese3` **erbt und erweitert z.B. die Klasse** `Activity`**, die im Package** `android.app` **der Bibliothek** `Android 3.0` **enthalten ist.**

Im Ordner `Assets` können Dateien beliebigen Inhalts abgelegt werden, auf die später per Dateinamen aus der Anwendung heraus, aber nicht aus anderen Anwendungen heraus, zugegriffen werden kann. Diese Dateien werden während des Paketierungsvorgangs, wenn unser Projekt erstellt und in eine `.dex`-Datei verpackt wird, ohne weitere Optimierung einfach 1:1 übernommen.

Im Gegensatz zu Ressourcen sind die Assets nicht konfigurationsabhängig organisierbar.

INFO

Was das konkret bedeutet betrachten wir im Abschnitt über das Ressourcensystem. Es sollte aber bereits hier erwähnt werden, dass das Ressourcensystem aus Gründen der Flexibilität in der Regel den Assets vorzuziehen ist.

Im Ordner `res` werden genau diese Ressourcen organisiert, die Struktur schauen wir uns später im Detail an.

Die Ressourcen werden ebenfalls durch den Paketierungsvorgang in das `.dex`-File überführt, allerdings werden für alle Ressourcen auch Identifizierer als ganzzahlige Konstanten in der `R.java` abgelegt. Auf die Ressourcen wird also nicht über den Namen zugegriffen, sondern über diese Konstante.

Darüber hinaus können die Ressourcen auch konfigurationsabhängig organisiert werden, z.B. Layouts speziell für Querformat und Zeichenketten in Deutsch und Englisch.

Das `AndoidManifest.xml`-File schließlich bildet das Herzstück für alle Vereinbarungen (deshalb Manifest), die unsere Anwendung trifft. Einige Angaben werden durch den Erstellungsassistenten hier bereits generiert, die Bearbeitung findet innerhalb der Eclipse mit einem speziellen Editor statt.

3.3 Die Android-Architektur

Android ist ein *Software-Stack* (Softwarestapel) für mobile Geräte. Unter einem *Software-Stack* versteht man die Gesamtheit aller Softwarekomponenten, die zur Verfügung gestellt werden (müssen), um eine bestimmte wohldefinierte Funktionalität zu realisieren.

Der übergeordnete Begriff dazu lautet *Solution-Stack*. Der *Solution-Stack* umfasst nicht nur die Softwarekomponenten, sondern ggf. auch Hardwarekomponenten, Standards und Konzepte zur Realisierung einer Funktionalität, der Lösung einer Aufgabe oder der Bereitstellung von Anwendungen und Diensten.

Der Begriff *Stack* (Stapel) bezeichnet sehr schön den modularen Aufbau von Systemen oder Lösungen. Die einzelnen Komponenten werden dabei übereinandergestapelt, wobei die tieferen Schichten die nötige Grundfunktionalität für die höheren Schichten bereitstellen. Jede einzelne Schicht kann wiederum aus verschiedenen Bausteinen bestehen, die unterschiedliche Funktionalitäten anbieten.

Je feiner die Schichten des Stapels unterteilt und je modularer jede einzelne Schicht aufgebaut ist, umso leichter wird es einzelne Schichten zu verändern oder um neue Module zu erweitern. Das erfordert aber eine saubere Planung der Schnittstellen (egal ob Hard- oder Software), über die die Module und Schichten miteinander kommunizieren können.

Android ist, meiner Meinung nach, extrem modular aufgebaut und besitzt gerade in der *Application Framework*-Schicht sehr sauber definierte Klassen und Schnittstellen. Durch die konsequente Modularisierung bis hin zur Anwendungsebene ist es z.B. auch möglich, die sogenannten *Home-Screens* und andere Standardanwendungen wie Mail, die Bildgalerie etc. durch andere Anwendungen zu ersetzen.

Wir wollen uns im Folgenden die Schichten des Android-Stacks genauer betrachten und an den Stellen, an denen es sinnvoll erscheint, bereits einen Hinweis darauf geben, welche *Klassen* wir im sogenannten Application Framework nutzen können, um auf Dienste der jeweiligen Schicht oder Komponente zugreifen zu können.

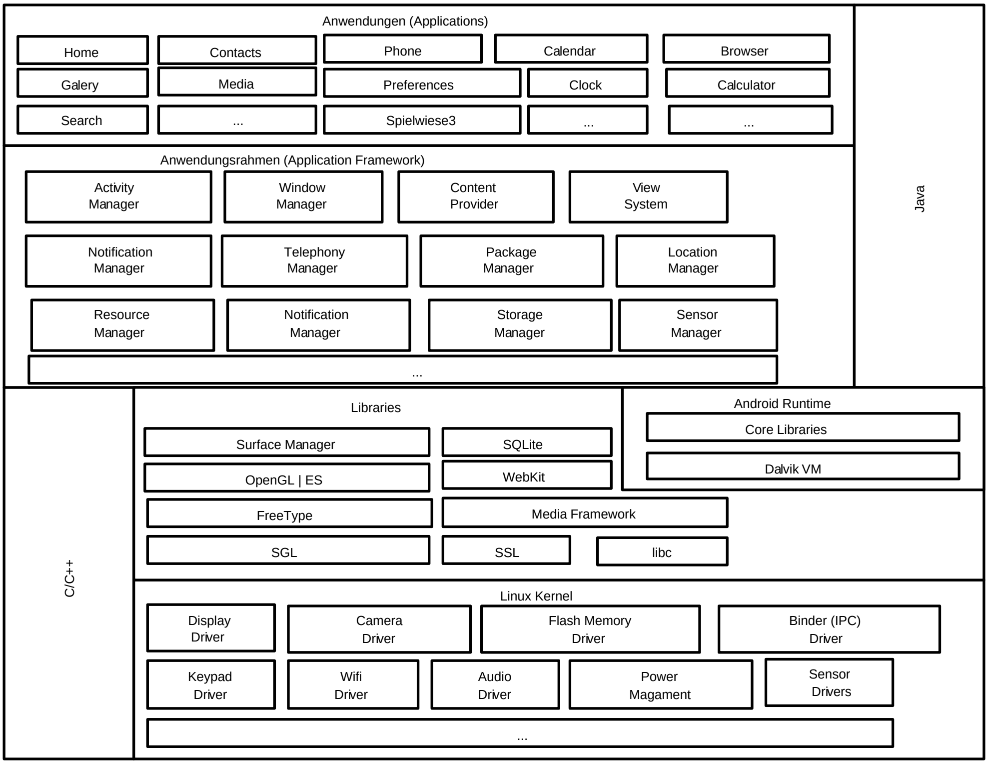


Abbildung 3.4: **Android Software-Stack**

In Kapitel 1 haben wir besprochen, woraus sich ein Android-System laut Googles Vorgaben zusammensetzen muss. Hier erkennen wir z.B. in der Anwendungsschicht die Standardanwendungen wieder, die ein Android-System von Haus aus mitbringt.

3.4 Allgemeine Grundlagen

Um auf Basis des Application Frameworks Anwendungen zu entwickeln müssen wir uns kurz mit einigen Konzepten und Begrifflichkeiten des Frameworks und der Laufzeitumgebung, vornehmlich der Core Library, auseinandersetzen. Wie in der Beschreibung des Software-Stacks zu erkennen ist besteht das Framework hauptsächlich aus Managern, die die Kommunikation mit den tieferen Schichten des Stacks und die Kommunikation innerhalb des Frameworks übernehmen. Über die Manager kommunizieren Anwendungen untereinander, stellen Anwendungen bestimmte Dienste zur Verfügung und lassen sich Dienste anderer Anwendungen und der Systembibliotheken nutzen.

Wenn ein Betriebssystem eine Anwendung startet, so geschieht das in der Regel über sogenannte *Eintrittspunkte* (Entry Points). Eintrittspunkte sind durch die Spezifikation festgelegte besondere Adressen innerhalb einer Anwendung, die aufgerufen werden sobald eine Anwendung (oder allgemein gesagt ein Modul) in den Speicher geladen worden ist und zur Ausführung gebracht werden soll. Technisch gesehen springt das Betriebssystem nach dem Laden an die entsprechende Adresse, der ausführende Prozessor setzt die Ausführung der Befehle an eben jener Stelle fort, und die Anwendung, das Modul, beginnt sozusagen mit der Ausführung seiner Aufgaben.

Das ist natürlich sehr einfach ausgedrückt – alle systemnahen Insider mögen mir die Vereinfachung verzeihen –, denn die Kontrolle verbleibt bei modernen Betriebssystemen im Grunde beim *Prozess-Scheduler* (dem Steuerprogramm) des Betriebssystems. Sie sorgt dafür, dass die Abarbeitung der Befehle zwischen den verschiedenen quasi gleichzeitig aktiven Modulen und Anwendungen »gerecht« aufgeteilt wird (Multitasking). Der Scheduler behält die Kontrolle dadurch, dass durch bestimmte Mechanismen die aktuelle Ausführung unterbrochen wird und der Prozessor quasi zum Scheduler zurückspringt. Diese Unterbrechung laufender Prozesse kann auf zweierlei Arten realisiert werden:

1. Kontrollierte, freiwillige Unterbrechung durch die Anwendung selbst. Dieser Mechanismus wird als kooperatives Multitasking bezeichnet, da der Scheduler nur dann zum Zuge kommt, wenn der aktive Prozess freiwillig durch den Aufruf von Betriebssystemfunktionen die Kontrolle an das Betriebssystem zurückgibt.

2. Kontrollierte zeitliche oder ereignisgesteuerte Unterbrechung durch die Schaltungslogik/Hardware. Dieser Mechanismus wird präemptives Multitasking genannt. Dabei wird durch eine hardwareseitige Unterbrechung (Interrupt) z.B. per Zeitgeber die Ausführung an den Scheduler zurückgegeben und der aktive Prozess unterbrochen, sodass der Scheduler die Ausführung an einen anderen Prozess weiterreichen kann.

Heute findet man in den Betriebssystemen in der Regel nur noch das präemptive Multitasking. Beim kooperativen Multitasking ist das System explizit auf die Kooperation der Programme angewiesen. Sollte ein Programm die Kooperation verweigern, können damit andere Prozesse lahmgelegt und im schlimmsten Falle das gesamte System zum Stillstand gebracht werden. Beim präemptiven Multitasking ist diese Gefahr relativ gering, da die Unterbrechung durch die Hardware gesteuert wird und eine Kooperation nicht mehr nötig ist. Inwieweit Prozesse »lahmgelegt« und andere bevorzugt werden, hängt lediglich von der Zuteilungsstrategie des Schedulers ab.

Allerdings ist es heutzutage gerade bei Anwendungen, die mit dem Benutzer über eine Benutzerschnittstelle interagieren, so, dass diese Anwendungen sich selbst weiterhin kooperativ geben und *ereignisgesteuert* aufgebaut sind. Das bedeutet, dass in der Anwendung z.B. keine Schleifen zur Entgegennahme von Tastendrücken realisiert sind, sondern die Anwendung auf solche Ereignisse reagiert sobald das Betriebssystem ein solches Ereignis erkennt.

Die Ereignissteuerung wird häufig so realisiert, dass eine Anwendung in eine Ereignisschleife eintritt und die ihr zugewiesene Nachrichtenwarteschlange regelmäßig abfragt. Dadurch ist die Anwendung wieder kooperativ, da die Nachrichtenwarteschlange vom Betriebssystem bereitgestellt wird und die Abfrage wiederum über Betriebssystemfunktionen stattfindet. Die Abarbeitung der Nachrichtenwarteschlange fällt wiederum unter das präemptive Multitasking, sodass die verschiedenen Prozesse nacheinander ihre Nachrichten abrufen und darauf reagieren können. Windows-Anwendungen arbeiten z.B. mit einer Nachrichtenwarteschlange, die in einer Nachrichtenschleife abgearbeitet werden. Das ist schon seit den frühen Versionen so, sogar noch zu Zeiten des kooperativen Multitaskings, und es ist auch heute noch so in Verbindung mit präemptiven Multitasking (seit Windows NT). Diese Art und Weise findet sich auch nahezu in allen Fenstersystemen oder Frameworks für Anwendungen mit fensterbasierten Benutzerschnittstellen.

Eine andere Realisierung ist, dass die Programme wiederum Eintrittspunkte für verschiedene Ereignisse definieren, die bei Eintreffen des Ereignisses durch das Betriebssystem angesprungen werden. Das ist ein Mechanismus, der in Android sehr intensiv genutzt wird. Der Entwickler hat selbst mit der Nachrichtenverarbeitung nichts mehr direkt zu tun, er definiert einfach die entsprechenden Eintrittspunkte und kann in der Applikation auf die Ereignisse gezielt reagieren.

Ereignisse und Nachrichten können aber nicht nur durch das Betriebssystem generiert werden, sondern auch von den Anwendungen und Modulen selbst. Dadurch ist dann wiederum eine Kooperation zwischen Anwendungen möglich, da über das Betriebssystem Nachrichten zwischen den Anwendungen ausgetauscht werden können.

Ein weiterer Vorteil, der sich aus dem präemptiven Multitasking in Zusammenhang mit der ereignisgesteuerten Architektur ergibt, ist, dass »wildgewordene« Programme, die z.B. in einer langen Verarbeitung fest hängen und nicht auf Benutzereingaben reagieren oder keine Bildschirmausgabe mehr generieren, erkannt und durch das Betriebssystem ggf.

beendet werden können. In Android erscheint in solchen Fällen der ANR-(Application not responding)-Dialog.

Wichtig ist in diesem Zusammenhang: Anwendungen und Module müssen definierte Eintrittspunkte bereitstellen, über die sie gestartet werden und über die sie auf bestimmte Ereignisse reagieren können.

Betrachten wir uns in diesem Zusammenhang noch einmal den Android-Software-Stack. Der Stack basiert auf einem Linux-Kernel. Das ist das eigentliche Betriebssystem, hier verstecken sich z.B. der Lader, der (native) Anwendungen zur Ausführung bringt, und der Scheduler, der für die Verteilung der Ausführungszeit sorgt.

Über dem Kernel ist die Android-Laufzeitumgebung angesiedelt. Die Laufzeitumgebung stellt quasi das Betriebssystem für die Android-Anwendungen bereit. In der Laufzeitumgebung verbirgt sich mit der Dalvik Virtual Machine der Prozessor, der den Android-Code des Application Frameworks und der Application-Schicht ausführt. Also: Linux stellt das grundlegende Betriebssystem bereit, das die Treiber, den Lader und den Scheduler für die nativen Anwendungen und Module zur Verfügung stellt. Das Betriebssystem und alle nativen Module werden direkt auf der CPU des Android-Gerätes ausgeführt. Ein Bestandteil dieser nativen Module ist die Android-Laufzeitumgebung. Diese sorgt dafür, dass für jede Android-Anwendung ein eigener Laufzeitprozess (auf dem Linux-Kernel) gestartet wird und ein eigener Dalvik-Prozessor zur Ausführung kommt. Dem Dalvik-Prozessor wiederum übergibt die Laufzeitumgebung die Android-Anwendung, die auf diesem virtuellen Prozessor ausgeführt wird. Der Linux-Kernel sorgt nun dafür, dass die einzelnen Android-Prozesse entsprechende Rechenzeit zugeteilt bekommen, stellt die Schnittstellen zu den nativen Funktionen (Bildschirmausgabe, Speicherzugriff, Netzwerk, Telefonie etc.) bereit und übergibt Ereignisse und Nachrichten (Eingaben, Sensorwerte etc.) an die Android-Laufzeitumgebung. In den Android-Anwendungen sind nun Eintrittspunkte definiert, die auf die entsprechenden Ereignisse und Nachrichten reagieren.

Wir haben nun des Öfteren über Anwendungen und Module gesprochen. Unter Anwendung verstehen wir hier die Gesamtheit von Modulen, die eine bestimmte Funktionalität bereitstellen und oft auch eine Benutzerschnittstelle haben. Ein Modul ist wiederum ein Stück Software (könnte auch Hardware sein) innerhalb des Solution-Stacks, das nur bestimmte Teilaspekte des Gesamtsystems anbietet und erst im Zusammenspiel mit anderen Modulen zu einer Anwendung wird. Das Modul definiert eine Funktionalität, die es anbietet, und Schnittstellen für andere Module um auf diese Funktionalität zuzugreifen.

Auch eine Anwendung kann selbst wieder ein Modul sein und in einem größeren Zusammenhang benutzt werden.

Im Kontext müssen wir noch den Begriff der Komponente einführen. Nach dem Aufkommen der objektorientierten Programmierung (und Android-Anwendungen werden objektorientiert in Java programmiert) hat sich der Begriff der komponentenbasierten Programmierung etabliert. Die objektorientierte Programmierung führt mit dem Begriff der Klasse. Eine Klasse kapselt die Realisierung einer Idee oder eines Konzepts (z.B. die Idee des Sen-

sors, der Messwerte liefert). Die Klasse stellt definierte Methoden zur Verfügung, um die gekapselte Idee zu benutzen, wie die Idee ausgeführt wird, d.h. die Implementierung, bleibt in der Klasse verborgen. Über ein »reales« Objekt der Klasse können wir das durch die Klasse eingeführte Konzept in unserer Anwendung nutzen. Klassen können wiederum ihre Eigenschaften vererben, Ideen und Konzepte können so erweitert oder bestimmte Aspekte der Idee mit neuen oder spezifischen Möglichkeiten realisiert werden (Spezialisierung).

Die Klasse beschreibt aber nur ein allgemeines Konzept. Eine Komponente ist eine konkrete, in sich abgeschlossene Umsetzung einer definierten Funktionalität, die als Klasse oder Klassenhierarchie ausgeführt ist und die wir ohne eigene Anpassung für unsere Zwecke nutzen können. Über die Schnittstellen kleben wir die Komponenten zusammen und entwerfen so unsere Anwendungen.

Eine weitere Möglichkeit die sich durch den objektorientierten und komponentenbasierten Ansatz ergibt, ist, bestehende Komponenten komplett durch andere Komponenten zu ersetzen, die neue Komponente muss sich lediglich an den Vertrag halten, also die Funktionalität und Schnittstellen zur Verfügung stellen, die von dieser Komponente erwartet werden. Ein Beispiel dafür ist z.B. die Oberfläche von Android-Geräten. Das gesamte System ist aus Komponenten aufgebaut, also auch die Screens, die sich uns nach dem Einschalten präsentieren. Einige Hersteller gehen nun her und tauschen die Standardkomponente für die Oberfläche durch eigene Komponenten aus, z.B. liefert HTC die Android-Geräte mit der Oberfläche HTC Sense aus, die sich etwas anders als die Standardoberfläche verhält und etwas anders aussieht.

Android definiert innerhalb des Application Frameworks eine Klassen- und Schnittstellenhierarchie, die wir zum Zugriff auf bestimmte Module des Software-Stacks nutzen können und definiert Komponenten, die in das Framework eingebunden und durch uns mit Leben gefüllt werden können: Diese Komponenten sind die Applikationskomponenten. Und mit diesen Komponenten wollen wir uns nun eingehender beschäftigen.

3.5 Grundlegende Eigenschaften von Android-Applikationen

Android-Applikationen werden in der Programmiersprache Java geschrieben. Zurzeit wird die Sprachversion Java 6 unterstützt. Der Java-Code wird durch den Java-Compiler kompiliert und durch das dx-Tool in den Byte-Code der Dalvik Virtual Machine übertragen. Der kompilierte Code sowie alle relevanten Daten- und Ressourcendateien werden in ein Android Package verpackt. Das Android Package ist eine Datei, die genau eine Anwendung enthält.

Das Android Package wird entweder über den Android-Market (oder alternative Plattformen) oder direkt mittels des Synchronisierungstools auf einem Android-Gerät installiert.

Jede Anwendung wird auf dem Gerät in ihrer eigenen virtuellen Sandbox ausgeführt. Das Android-Betriebssystem basiert auf einem mehrbenutzerfähigen Linux-System. Mehr-

benutzersysteme führen die Anwendungen unterschiedlicher Nutzer in eigenen isolierten Prozessen aus. Für die einzelnen Nutzer stellt sich das System so dar, als würde nur er alleine das System nutzen. Die Anwendungen des einen Nutzers wissen grundsätzlich erst einmal nichts über die Anwendungen anderer Nutzer und können auch nicht untereinander direkt kommunizieren. Somit kann ein Benutzerprozess auch nicht auf Daten eines anderen Prozesses zugreifen oder die Ausführung von Anwendungen stören.

Weiterhin werden in einem echten Mehrbenutzersystem alle Systemobjekte wie Dateien, Geräte etc. einem oder auch mehreren Benutzern mit unterschiedlichen Rechten zugewiesen. Das geschieht über eine eindeutige ID, die den Benutzer innerhalb des Systems ausweist, und durch verschiedene Attribute, die den Zugriff auf ein Objekt regeln. Die unterschiedlichen Objekte innerhalb des Mehrbenutzersystems haben auch einen Besitzer, das ist in der Regel der Benutzer, der ein Objekt erstellt hat. Der Besitzer, und eigentlich nur der Besitzer, kann für seine Objekte entscheiden, ob er anderen Benutzern Rechte auf diese Objekte gewährt.

ACHTUNG

Das ist ein fundamentales Konzept, um eine hohe Sicherheit von Anwendungen zu gewährleisten. Wenn, ja wenn es nicht so wäre dass es fast immer einen privilegierten »Benutzer« bzw. Prozess geben muss, der umfassende Rechte besitzt, um die Betriebssystemaufgaben selbst wahrzunehmen, z.B. um überhaupt Rechte gewähren und Dateien anlegen zu können. Diese privilegierten Benutzer sind häufig das Einfallstor für Schadprogramme. Wenn es durch welchen Umstand auch immer gelingt, eine Anwendung unter einem privilegierten Nutzer oder auch nur als fremden Nutzer einzuschleusen, kann diese Anwendung natürlich auf Objekte zugreifen auf die sie das gar nicht soll.

Was bedeutet das nun im Zusammenhang mit unserem Android-Gerät? Im Gegensatz zu einem Linux-Host, der über Terminals mehrere Benutzer bedient, wird ein Android-Gerät niemals gleichzeitig von mehreren Benutzern genutzt.

Android nutzt daher die Mehrbenutzerfähigkeit dazu, jede Applikation als eigenen Benutzer in einem eigenen Prozess laufen zu lassen. Das Betriebssystem weist bei der Installation jeder Applikation eine eindeutige Benutzer-ID zu und setzt die Berechtigungen für alle Objekte der Anwendung für diese und nur für diese Benutzer-ID. Dadurch werden alle Applikationen erst einmal grundsätzlich voneinander abgeschottet, und dieser Mechanismus ist die grundlegende Basis für das Sicherheitssystem im Android-Betriebssystem. Eine Anwendung kann nur auf die eigenen Objekte und nicht so ohne Weiteres auf die Objekte anderer Anwendungen, die eine andere Benutzer-ID haben, zugreifen.

Dieses Prinzip nennt man *das Prinzip der geringsten Privilegien* (*Principle of least privilege*), die Anwendung hat nur Zugriff auf die Komponenten, die sie benötigt und kein einziges weiteres Privileg mehr. Da alle Systemkomponenten wie Telefonie, Sensoren, Datenspeicher, GPS, Kamera etc. der Android-Laufzeitumgebung selbst wieder Anwendungen sind bzw. durch die Laufzeitumgebung gesteuert werden, heißt das aber auch, dass eine Anwendung grundsätzlich erst einmal keinen (unerwünschten) Zugriff auf diese Komponenten hat.

Was geschieht nun aber, wenn eine Anwendung auf Systemkomponenten oder Daten oder Dienste anderer Anwendungen zugreifen muss? Wie bereits beschrieben, kann immer nur der Besitzer eines Objekts den Zugriff darauf gewähren. In Android kommt damit dem Manifest (dem »Vertrag«) einer Applikation besondere Bedeutung zu. Neben einigen weiteren Vereinbarungen ist im Manifest auch festgeschrieben, auf welche Komponenten eine Anwendung zugreifen möchte. Das Manifest wird vom Paketmanager, der auch die Installation von Anwendungen übernimmt, vor der Installation auf die gewünschten Privilegien untersucht und der Besitzer kann nun entscheiden, ob er der Anwendung Zugriff auf die Komponente gewähren will.

Und das Gute daran ist, der Besitzer, das sind tatsächlich wir, die wir das Gerät in der Hand haben. Es ist also kaum möglich dass sich eine Anwendung unerwünschte Zugriffe erschleicht.

ACHTUNG

Wenn, ja wenn es nicht so wäre, dass es immer wieder einen findigen Tüftler gibt, der im Betriebssystem oder in der Laufzeitumgebung eine Lücke findet, um das eigentlich sichere System auszuhebeln. Der Vorteil eines Open-Source-Systems ist hier aber, dass diese Lücke theoretisch von jedermann erkannt und auch, wenn er denn freundlich gesonnen ist, geschlossen oder zumindest veröffentlicht werden könnte.

Wenn wir also nicht wollen, dass eine Anwendung unsere Standortdaten verarbeitet und/oder Zugriffe auf das Netzwerk ausführt oder SMS verschickt, dann verweigern wir bereits bei der Installation die Erlaubnis dafür.

INFO

Wenn wir eine Applikation installieren, bedeutet die Verweigerung der Erlaubnis aber auch, dass wir die Applikation überhaupt nicht installieren. Es ist nicht möglich, nur einzelne Berechtigungen zu gewähren. Das heißt, wir müssen die Applikation entweder als Ganzes nutzen oder gar nicht. Das ist auch sinnvoll, denn ansonsten würde das Sicherheitssystem viel komplizierter werden und damit die Gefahr von Sicherheitslücken ansteigen. So muss während der Ausführung nicht geprüft werden, ob ein Recht vergeben wurde oder nicht, die Prüfung findet schon vorher statt und man holt sich keine unerwünschte Funktion auf das Gerät.

Gewähren wir den Zugriff, dann wird die Benutzer-ID der Applikation mit den Objekten verknüpft und die entsprechenden Berechtigungen für die Benutzer-ID werden gesetzt, und die Anwendung kann auf die gewünschten Komponenten zugreifen.

Was bedeutet das für uns als Entwickler?

Zum einen bedeutet es, das wir nicht einfach drauflosgaloppieren und munter auf alles zugreifen können ,was uns unter die Finger kommt. Wir müssen uns also Gedanken machen, was unsere Applikation können soll und was sie für Zugriffe benötigt. Die Berechtigungen, die wir anfordern, müssen wir im Manifest entsprechend vereinbaren.

Zum anderen bedeutet es, diszipliniert vorzugehen. Man könnte ja nun auf die Idee kommen, einfach alles anzufordern was das Gerät so hergibt. Das ist allerdings keine wirklich gute Idee, da das Sicherheitskonzept von Android auf der Möglichkeit beruht zu entschei-

den, welche Funktionen man erlauben möchte, und ich muss als Anwender auch nachvollziehen können, warum die Anwendung eine bestimmte Berechtigung anfordert. Wenn unsere Applikation etwas wirklich Nützliches kann, aber ein Recht anfordert, das sie gar nicht benötigt, wird der Benutzer die Anwendung vielleicht nicht installieren, weil er genau diese Erlaubnis nicht erteilen möchte. Ich möchte z.B. kein Spiel installieren, das Zugriff auf GPS und SMS und das Netzwerk anfordert, obwohl diese Komponenten für das Spiel wahrscheinlich keinen wirklichen Sinn haben, außer ggf. Daten an den Hersteller zu schicken. Wir müssen also bei der Entwicklung auch immer entscheiden, wie wichtig eine Funktion für die Anwendung ist und ob es vielleicht sinnvoll sein kann, unterschiedliche Versionen anzubieten, z.B. eine Version, die GPS-Daten anfordern können muss, und eine, die das nicht benötigt.

Ein weiterer Aspekt kommt bei der Veröffentlichung der Applikation in einer Market-Plattform zum Tragen. Bei einigen sensiblen Funktionen wie dem Zugriff auf das Netzwerk, die GPS-Daten oder Telefonie sollten wir dem Anwender genau erklären, warum unsere Applikation genau diese Zugriffe benötigt. Dass eine Geotagging-Anwendung Zugriff auf die GPS-Funktionen benötigt, ist offensichtlich, wenn die Anwendung allerdings auch noch Netzwerkzugriff anfordert sollte dieser Aspekt genau erklärt werden, denn ich möchte wahrscheinlich nicht, dass meine Positionsdaten irgendwo hingeschickt werden. Wenn das Tagging-Modul aber Zugriff auf Google-Maps benötigt, um die Tracks anzeigen zu können, und dieser Umstand wird in der Beschreibung genau erklärt, dann könnte ich mich eher dazu bereit erklären, die Funktion zuzulassen.

INFO

Das hier Beschriebene ist vielleicht ein Knackpunkt bei der Verwendung von mobilen Geräten, die Standortfunktionen beinhalten und Netzwerkfunktionalität anbieten, und auch ein Nachteil der Apache-Lizenz, unter der große Teile von Android stehen. Denn die Applikationen selbst müssen nicht als Open-Source veröffentlicht werden, und das erschwert die Möglichkeit herauszufinden, was eine Anwendung nun wirklich mit einer Funktion anfängt. Wenn ich den Zugriff auf Standortfunktionen gewähre, muss ich tatsächlich das Vertrauen haben dass die Applikation diese Daten nicht irgendwohin schickt und keine (unerwünschten) Bewegungsprofile heimlich aufgezeichnet werden.

Kehren wir noch einmal zum Anfang zurück. Jede Anwendung läuft in einem eigenen, isolierten Benutzerprozess mit einer eigenen Benutzer-ID. Neben den oben diskutierten Sicherheitsaspekten hat dieser Umstand für das Betriebssystem einen weiteren Vorteil, der uns in der Entwicklung aber auch einige Sorgfaltspflichten auferlegt.

Das Betriebssystem kümmert sich darum, eine Anwendung in einem eigenen Prozess zu starten. Wie eine Anwendung gestartet wird, sehen wir später noch, wichtig ist, erst einmal nur zu wissen dass die Kontrolle vom Start der Anwendung weg beim Betriebssystem verbleibt. Dass mobile Geräte trotz ihrer Leistungsfähigkeit über stärker begrenzte Ressourcen als PCs verfügen, dürfte offensichtlich sein. Dass die Funktionsfähigkeit einiger grundlegender Dienste wie der Telefonie bei einem Smartphone nahezu 100% sichergestellt werden sollte, dürfte eine obligatorische Forderung sein. Da Anwendungen aber während ihrer Laufzeit Ressourcen wie Speicher, Strom und Rechenzeit verbrauchen, ist es wichtig,

dass das Betriebssystem stets die Kontrolle über die Anwendungen hat. Durch das Prozessmodell von Android kann das Betriebssystem jeden einzelnen Prozess unabhängig von den anderen Prozessen kontrollieren. Damit nimmt das Betriebssystem uns einige Arbeit ab, wir müssen keine Möglichkeit für das Beenden der Anwendung einbauen und uns auch nicht so sehr um den Speicherverbrauch kümmern, denn das Betriebssystem kann unsere Anwendung unterbrechen, wenn ein Telefonanruf eingeht, zur Anwendung zurückkehren, wenn wir aufgelegt haben, nicht mehr benötigten Speicher freigeben, die Anwendung schlafen legen, wenn wir zu einer anderen wechseln, und die Anwendung ggf. beenden und aus dem Speicher entfernen, wenn nicht genügend Ressourcen zur Verfügung stehen. Zur Not kann das Betriebssystem auch einen wild gewordenen Prozess, der z.B. nicht mehr auf Benutzereingaben reagiert, komplett aus dem Speicher werfen und damit die Funktionsfähigkeit des Geräts garantieren.

Das ist eine wirklich gute Sache und macht das System sehr stabil und flexibel, hat aber auch Auswirkungen auf die Entwicklung. Wir müssen im Grunde immer damit rechnen, dass unsere Anwendung unterbrochen und vielleicht sogar komplett aus dem Speicher entfernt wird. Das Lebenszyklusmodell für Anwendungen in Android berücksichtigt diesen Umstand und erlaubt uns, auf diese Ereignisse zu reagieren und den aktuellen Zustand unserer Anwendung z.B. zu speichern und später wieder herzustellen.

INFO

Es ist interessant, dass gerade auch die kleine Bildschirmgröße einer jener limitierenden Faktoren darstellt, der ein besonderes Prozess- und Lebenszyklusmodell notwendig macht. Im Gegensatz zu einem Computerbildschirm mit sagen wir ab 17" Bildschirmdiagonale aufwärts können auf der kleinen Fläche des mobilen Geräts wohl kaum Anwendungen in Fenstern dargestellt werden. Auf solchen Geräten werden eher sog. Z-Fenstersysteme benutzt, bei denen sich die Anwendungsoberflächen übereinander stapeln. Hier bewegt man sich nicht frei durch die Liste der offenen Fenster, sondern durch den Stapel wieder zurück oder auch immer wieder zu einer neuen Anwendung, die sich über die anderen Anwendungen legt. Wenn man dieses Bild vor Augen hat, ist es verständlich, dass das Betriebssystem eine viel stärkere Kontrolle über die laufenden Anwendungen ausüben muss und z.B. Anwendungen, die weiter unten im Stapel liegen einfach komplett aus dem Speicher zu entfernen und erst wieder zu aktivieren, wenn sie wieder an der Reihe sind.

3.6 Organisation von Android-Anwendungen

3.6.1 Das Android Package

Android-Anwendungen werden in Java geschrieben, die dann in den Bytecode für die Dalvik-VM übersetzt werden. Zusammen mit weiteren Dateien (Ressourcendateien, dem Manifest, Datendateien) werden die übersetzten Dateien in einem *Android Package File* zusammengefasst. Das Android Package File ist ein Archiv-File (das Format entspricht einem ZIP-Archiv, ähnlich wie die JAR-Packet-Dateien in Java) und wird mit dem Suffix `.apk` benannt (z.B. `Scrapbook.apk`). Diese Datei beinhaltet alle Dateien und den Code, der genau eine Anwen-

dung (Application) ausmacht. Anwendungen werden einfach mittels dieser Datei über den Android-Market oder anderweitige Möglichkeiten auf einem Gerät installiert.

Ein Android Package sieht typischerweise wie folgt aus:



Abbildung 3.5: **Struktur des .apk-Files (entpackt mit einem ZIP-kompatiblen Packprogramm)**

Das Paket wird während des Erstellungsprozesses erzeugt. Unter dem Ordner `META-INF` stehen Metadaten und die Zertifikatsinformationen ,mit denen die Anwendung signiert ist.

Wir erkennen sehr schön die Ressourcenstruktur, die ein Abbild der Struktur des Projekts ist. Allerdings werden beim Erstellen der Ressourcen bestimmte Optimierungen durchgeführt, so sind die XML-Dateien keine Textdateien mehr, sondern binäre, optimierte Darstellungen des XML-Dokuments. Genauso verhält es sich mit dem Manifest.

Unsere kompilierte Anwendung selbst steckt nun in der Datei `classes.dex`, die einen optimierten Dalvik-Bytecode enthält, der auf der Dalvik-VM zur Ausführung gelangt.

Wenn das Paket über den Market veröffentlicht wird, dann schaut der Market bereits in das Paket rein und ermittelt aus dem Manifest den Ziel-API-Level sowie weitere Vereinbarungen über unsere Anwendung. Anhand dieser Informationen kann unsere Anwendung dann im Markt gefiltert werden.

Darüber hinaus wird unsere Anwendung mittels des enthaltenen Zertifikats unserem Entwickler-Account zugeordnet, das heißt es besteht »Klarheit« über den Publisher.

3.6.2 Das Manifest

Anwendungen benötigen einen Eintrittspunkt, an dem die Ausführung des Programms beginnt. In den Hochsprachen wie Java oder C/C++ sind das bestimmte Funktionen oder Methoden.

Der Eintrittspunkt für C/C++ Programme ist in der Regel die Funktion `main()`:

```
int main(int argc, char* argv)
{
    //... Tu was
    return 0; //Beenden des Programms
}
```

Wenn das Programm durch den Compiler übersetzt wird, wird für die Funktion `main()` ein entsprechender Code erzeugt, damit das Betriebssystem nach dem Laden des Programms an diese Stelle springen kann.

Anwendungen für Windows, die in C/C++ geschrieben werden, haben einen Einsprungpunkt namens `WinMain()`, dynamische Bibliotheken unter Windows den Eintrittspunkt `DLLMain()`.

In Java ist dieser Eintrittspunkt die statische Methode `main()` innerhalb einer Klasse:

```
package de.androidpraxis.JavaGrundlagen;
public class Start {
    public static void main(String[] args) {
    // ... Tu was
    }
}
```

Übergibt man das Programm an die Java VM, gibt man die entsprechende Klasse an (`Start`), und Java beginnt die Ausführung in der statischen Methode `main()`.

Diese Eintrittspunkte sind Bestandteil einer Festlegung, die bei der Definition der jeweiligen Sprache getroffen wurden und durch die Compiler bzw. Laufzeitumgebungen eingehalten werden.

Unter Android existiert kein solcher einzelner Eintrittspunkt und keine implizite Festlegung dafür. Je nachdem, welche Applikationskomponente wir verwenden, müssen wir Android mitteilen, welche Komponente durch welches Ereignis gestartet wird. Das bedeutet, dass wir die Festlegung treffen und der Laufzeitumgebung bekannt machen müssen.

Die Bekanntgabe unserer Festlegung erfolgt durch das *Manifest* in einer *Manifestdatei*.

Die Manifestdatei ist essenzieller Bestandteil jeder Android-Applikation. Per Definition heißt die Manifestdatei *AndroidManifest.xml* (und zwar genau so, Groß- und Kleinschreibung ist hier wichtig).

Installieren wir nun eine Anwendung auf dem Gerät, schaut die Laufzeitumgebung als Erstes in diese Manifestdatei, um gewisse Festlegungen zu erfahren. Neben den Festlegungen, was wie wann aus welchem Grund gestartet werden soll, beschreibt die Manifestdatei unsere Applikation noch wesentlich detaillierter, z.B. auf welche Gerätekomponenten die Applikation zugreifen wird, ob sie den Telefoniedienst in Anspruch nimmt oder auf das Internet zugreift.

Bilden also die per Definition festgelegten Eintrittspunkte in C/C++ oder Java den Einstieg in die jeweilige Anwendung, so schaut Android zuerst in das Manifest, um alles über unsere Absichten zu erfahren.

Die Manifestdatei ist eine XML-Datei, die einem wohl definierten Schema folgt, das bedeutet, die Elemente und die Struktur der Datei sind festgelegt und nicht durch eigene Elemente erweiterbar.

An dieser Stelle schauen wir uns kurz die Struktur des Manifests an, ohne im Detail auf die Elemente einzugehen, das erledigen wir später wenn es »ans Eingemachte« geht.

ABSCHNITT	BEMERKUNG
<?xml version="1.0" encoding="utf-8"?>	XML Deklaration (obligatorisch)
<manifest>	Leitet das Manifest ein (obligatorisch)
<uses-permission/>	Legt fest, welche *permissions* (Genehmigung) die Anwendung benötigt. Wenn der Anwender der Applikation die Genehmigung nicht erteilt, schlagen die Zugriffe auf die entsprechenden Komponenten fehl.
<permission/>	Mit diesem Element kann man eigene Genehmigungen festlegen, um Funktionen oder Daten in der eigenen Anwendung zu schützen.
<permission-tree/>	
<permission-group/>	
<instrumentation/>	
<uses-sdk/>	Angaben zur Version des SDK, das die Anwendung nutzt. Anwender, die ein Gerät haben, das nicht mit der benötigten Android-Version kompatibel ist, bekommen die Applikation im Android-Market nicht angezeigt bzw. können diese nicht installieren.
<uses-configuration/>	Legt fest, welche Eingabegeräte die Anwendung benötigt (z.B. eine echte Tastatur, welche Navigationstasten etc.). Anwender, die ein Gerät haben, das die benötigten Features nicht bietet, bekommen die Applikation im Android Market nicht angezeigt bzw. können diese nicht installieren.
<uses-feature/>	Legt fest, welche Hardwarekomponenten (z.B. Kamera, Mikrofon, Bluetooth) oder Software-Features (Live Wallpaper, SIP/VoIP) die Anwendung benötigt. Anwender, die ein Gerät haben, das die benötigten Features nicht bietet, bekommen die Applikation im Android-Market nicht angezeigt bzw. können diese nicht installieren.
<supports-screens/>	Gibt an, welche Bildschirmabmessungen durch die Anwendung unterstützt werden.

Tabelle 3.1: **Aufbau der AndroidManifest.xml Datei**

ABSCHNITT	BEMERKUNG
<application>	Konfiguration der einzelnen Anwendungskomponenten
<activity>	Konfiguration von Activity-Komponenten. Eine Activity ist eine Komponente, die mit dem Benutzer interagiert und immer eine Benutzeroberfläche besitzt.
<intent-filter>	Spezifizieren der *Intents* (grob Übersetzt in diesem Kontext *Absichten* oder *Zweck*), auf die die Activity reagieren soll. Die *Intents* sind der Dreh- und Angelpunkt für das Auslösen von Funktionen innerhalb der Anwendungen, und das Starten einer Anwendung (bzw. einer Aktivität innerhalb der Anwendung) ist nur eine Absicht, ein Zweck unter vielen anderen Intents. Mit den *Intents* definiert man *Ereignisse*, auf die die Aktivität reagiert.
<action/>	
<category/>	
<data/>	
</intent-filter>	
<meta-data/>	Zusätzliche Name-Wert-Pärchen, die von der Komponente abgefragt werden können.
</activity>	
<activity-alias>	Legt Alias-Namen für Aktivitäten fest.
<intent-filter></intent-filter>	
<meta-data/>	
</activity-alias>	
<service>	Konfiguration von *Services* (*Diensten*). *Services* sind Komponenten, die kein Benutzerinterface anbieten und nicht direkt mit dem Benutzer interagieren, Dienste laufen im Hintergrund ab. Ein prominentes Beispiel ist der Mediaplayer, der Musik im Hintergrund abspielt.
<intent-filter></intent-filter>	
<meta-data/>	
</service>	

Tabelle 3.1: **Aufbau der AndroidManifest.xml Datei (Forts.)**

ABSCHNITT	BEMERKUNG
<receiver>	Konfiguration von *Broadcast Receivern* (*Rundruf-Empfängern*). *Boadcast Receiver* sind Komponenten, die auf *Intents* reagieren können, die vom Betriebssystem oder von anderen Anwendungen gesendet werden. Andere Komponenten (wie eine *Activity* oder ein *Service*) der Anwendung müssen (noch) nicht laufen, um die Nachricht zu empfangen und darauf zu reagieren.
<intent-filter></intent-filter>	
<meta-data/>	
</receiver>	
<provider>	Konfiguration von *Content-Providern* (*Inhalts-anbietern*). Durch *Content-Provider* kann die Anwendung anderen Anwendungen Zugriff auf in ihr gespeicherte Daten ermöglichen
<grant-uri-permission/>	
<meta-data/>	
</provider>	
<uses-library/>	Konfiguration von *Bibliotheken* (*Shared Libraries*), die die Anwendung benötigt. Alle Bibliotheken aus dem *Android Package* werden automatisch zur Anwendung geladen, Bibliotheken von Drittanbietern wie z.B. *maps* (Google Maps-API) müssen hier explizit aufgeführt werden.
</application>	
</manifest>	

Tabelle 3.1: **Aufbau der AndroidManifest.xml Datei (Forts.)**

Glücklicherweise stellt das ADT-Plug-in (Application Development Tools) für Eclipse einen Editor für das Manifest zur Verfügung, so dass wir in der Regel nicht direkt in der XML-Datei arbeiten müssen.

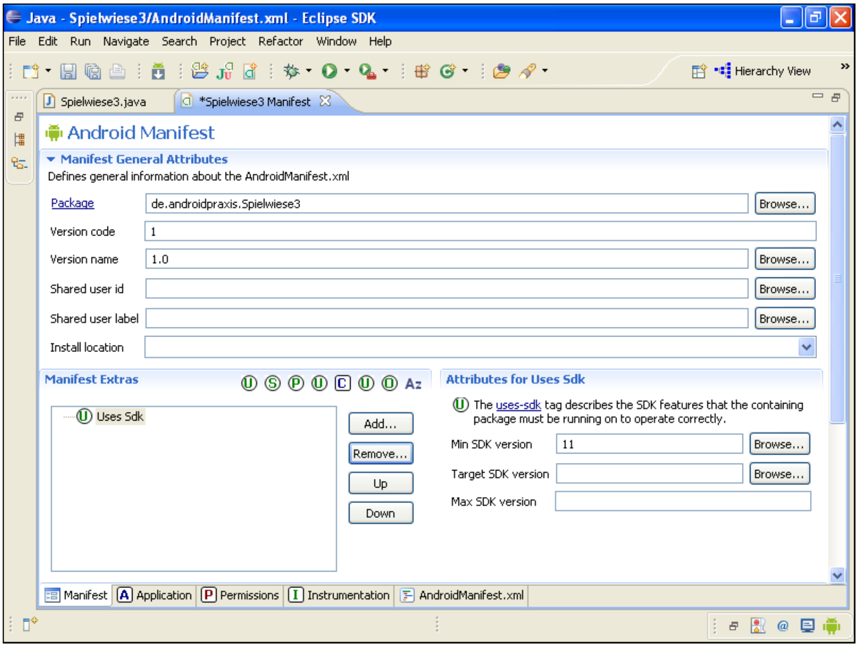


Abbildung 3.6: **Der Manifest Editor**

Hier sehen wir das Manifest zur Spielwiese. Wir erkennen den *Package Name* sowie die *Min SDK Version*, die wir im Erstellungs-Assistenten angegeben haben.

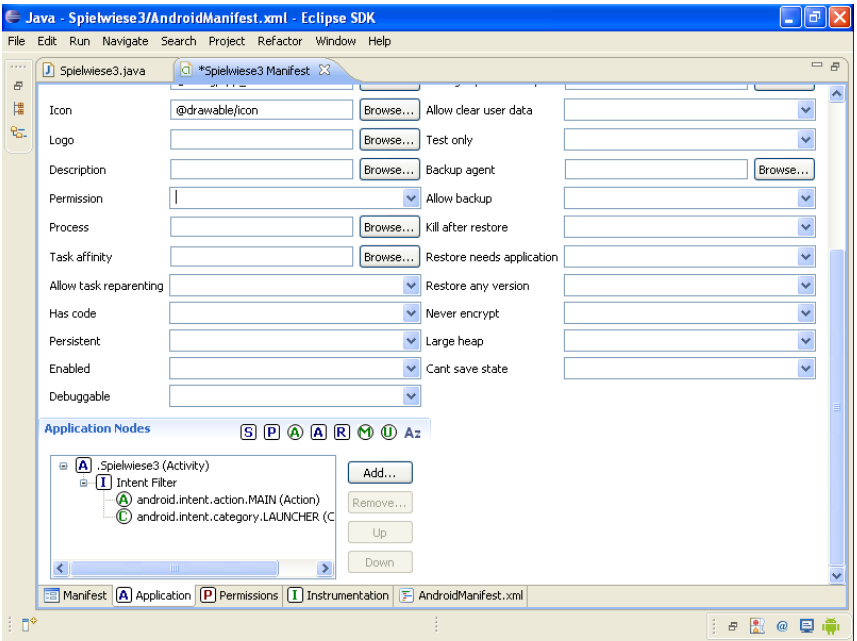


Abbildung 3.7: **Der Manifest-Editor – Einstellungen der Anwendung**

Auf dieser Seite sehen wir die Angaben zur Anwendung, ganz speziell die Konfiguration der *Activity*, die wir mit der Option *Create Activity* angelegt haben.

Wie die einzelnen Komponenten konfiguriert werden betrachten wir uns näher, wenn wir die einzelnen Komponenten durchsprechen.

3.7 Nachrichten und Ereignisse

Die Laufzeitumgebung von Android und das Application Framework sind, wie in modernen Betriebssystemen üblich, ereignisgesteuert ausgelegt. Das bedeutet, dass innerhalb einer Anwendung auf bestimmte Ereignisse reagiert und dadurch bestimmte Funktionen ausgeführt werden. Wird eine Anwendung zur Ausführung gebracht, so landet sie nach dem Durchlaufen bestimmter Startsequenzen unweigerlich in einer Warteschleife in der sie einfach auf Ereignisse wartet.

Außerdem haben wir bereits erkannt, dass das Android-System stark komponentenbasiert ausgeführt ist. Anwendungen können wiederum Komponenten sein, die eine bestimmte Funktionalität anderen Komponenten zur Verfügung stellen. Damit diese Funktionalität durch andere Anwendungen genutzt werden kann, müssen die (unabhängigen) Komponenten untereinander kommunizieren können. Ein Beispiel dafür wäre die Auswahl eines Kontakts aus dem Adressbuch. Die Adressbuchanwendung stellt diese Funktion zur Verfügung. Eine andere Anwendung muss mit der Adressbuchverwaltung kommunizieren können, um die Auswahl eines Kontakts anzufordern und den Kontakt zu übernehmen.

Um die Ereignisbehandlung und die Kommunikation zwischen den Komponenten in Android besser zu verstehen, müssen wir uns klar machen, um was für Ereignisse und Nachrichten es sich im Detail handelt. Es gibt Ereignisse unterschiedlicher Kategorien, was den Auslöser, ihre Herkunft und auch die Häufigkeit ihres Auftretens angeht.

Eine Unterteilung, die wir vornehmen können, ist:

1. Vom Benutzer ausgelöste Ereignisse
2. Durch die Hardware ausgelöste Ereignisse
3. Durch Software ausgelöste Ereignisse
4. Kommunikation durch Nachrichten zwischen Modulen

Ereignisse wie Tastendrücke oder Gesten auf dem Touchscreen werden durch die unmittelbare Interaktion des Benutzers mit der Hardware ausgelöst. Mittelbar können daraus wieder Ereignisse entstehen, die durch das Betriebssystem oder durch Anwendungen ausgelöst werden.

Unter Ereignissen, die direkt von der Hardware ausgelöst werden, verstehen wir im Zusammenhang mit den mobilen Geräten Ereignisse, die von den Gerätekomponenten erzeugt werden, wie z.B. die Änderung von Sensorwerten, Batteriestatusmeldungen, Änderungen an der Systemkonfiguration durch Anschließen von Kopfhörern oder das Einstecken des Geräts in eine Docking-Station.

Durch Software ausgelöste Ereignisse sind Ereignisse, die entweder in Reaktion auf Benutzereingaben wie die Auswahl von Menüpunkten, das Drücken von Navigationstasten oder das Aufnehmen eines Bildes entstehen oder durch z.B. Kalenderereignisse oder Benachrichtigungen.

Es ist offensichtlich, dass die benutzerintendierten Ereignisse wie Tastendrücke, Gesten etc. und die softwareintendierten Ereignisse in der Regel wenig zeitkritisch sind und auch nicht in schneller Folge eintreten. Im Gegensatz dazu treten Ereignisse, die von Sensoren ausgelöst werden, in der Regel sehr häufig und in schneller Abfolge auf.

Auch die Schichten innerhalb des Solution-Stacks, in denen die Ereignisse auftreten und der Weg, den sie zu den Empfängern nehmen, ist unterschiedlich. Ereignisse, die in der Software (den Anwendungen oder der Laufzeitumgebung) ausgelöst werden, werden innerhalb der Laufzeitumgebungsschicht bzw. innerhalb des Frameworks und der Anwendungsschicht verpackt, verschickt und bearbeitet. Hardwareereignisse treten in der untersten Schicht auf und erzeugen in der Regel Unterbrechungen (Interrupts) innerhalb des Kernels oder müssen durch den Kernel periodisch abgefragt werden (Polling). Der Kernel muss diese Ereignisse, z.B. die Sensorwerte, in die Laufzeitumgebung und darüber zu den Empfängern schicken.

Um die Ereignisse aus der Hardware- bzw. Betriebssystemschicht und der Laufzeitumgebung zu übermitteln muss ein Mechanismus gewählt werden, der einen geringen Overhead besitzt und der auch für Ereignisse in schneller Abfolge und hoher Anzahl geeignet ist.

Um das zu erreichen, wird die Behandlung solcher Ereignisse mittels sogenannter Listener abgewickelt. Listener sind seit Anbeginn der Java-Sprache dort das Mittel der Wahl, um auf Ereignisse zu reagieren. Grundlage für die Listener sind sogenannte Interfaces, das heißt Schnittstellendefinitionen. Eine Schnittstellendefinition dient dazu, die Vereinbarung darüber zu treffen, welche Methoden eine Komponente bereitstellen muss und wie die Parameter und Rückgabewerte der Methoden aussehen. Die Funktionalität selbst muss durch die Komponente realisiert werden. Mit diesem Mechanismus werden Listener gebaut. Um z.B. auf einen Tastendruck reagieren zu können, erwartet der Manager für Tastatureingaben, dass ein Empfänger die Schnittstelle `KeyEvent.Callback` implementiert. Der Manager für Sensoren erwartet, dass der Empfänger die Schnittstelle `SensorListener` implementiert. Wenn unsere Komponente nun also auf Sensorwerte reagieren soll, implementieren wir die entsprechende Schnittstelle und können darin unsere Funktionalität definieren, die etwas mit den Sensorwerten anstellt. Die Schnittstellen sind hier sehr schlank gehalten, ein Tastaturevent enthält z.B. nur den Code der gedrückten Taste und zusätzliche Infos, ob die Taste schon länger gedrückt wird, ein Sensorevent enthält nur die Sensorwerte. Damit ist ein umständliches ein- und auspacken der Ereignisinformationen nicht nötig.

Wenn wir nun also in unserer Komponente die entsprechende Schnittstelle implementiert haben, können wir diese Komponente z.B. beim Sensormanager als Listener registrieren. Activities (die ein Benutzerinterface haben) z.B. implementieren bereits die Schnittstelle für

Eingabeereignisse und werden beim Start durch das Laufzeitsystem ohne unser Zutun beim entsprechenden Manager registriert.

Die Manager selbst führen einfach eine Liste von Listenern und sind relativ eng mit der tiefer liegenden Betriebssystemschicht verzahnt. Wird durch den Kernel nun ein Ereignis an die Laufzeitumgebung übergeben, müssen die entsprechenden Manager lediglich die Listener durchlaufen und die entsprechende Methode (z.B. `onKeyDown`) mit dem jeweiligen Parameter (`keyCode`, `keyEvent`) aufrufen. Dieser Mechanismus geht sehr schnell und erlaubt auch die effiziente Verteilung einer großen Menge an Ereignissen, die in schneller Folge eintreten.

Es ist wichtig zu sehen, dass die Menge dieser Ereignisse durch das Gesamtsystem definiert ist und pro Ereignis bzw. Ereignisquelle eine Schnittstellendefinition für einen entsprechenden Listener existieren muss. Das bedeutet aber auch, dass bei einem Systemupdate, wenn neue Hardwarekomponenten mit neuen Ereignissen hinzukommen, oft auch neue Listener-Definitionen in die Laufzeitumgebung aufgenommen werden müssen, denn die Anwendungen müssen bereits bei der Entwicklung die entsprechenden Schnittstellen kennen. Man bindet die Anwendung also »früh« an bestimmte Funktionalitäten, und die Anwendung kann nicht auf Systemen laufen, die die Funktionalität nicht implementiert haben. Außerdem ist es offensichtlich, dass das Modul, das auf ein Ereignis reagieren soll, bereits zur Ausführung gebracht werden musste und damit auch Speicher belegt und Rechenzeit verbraucht.

Auf der Anwendungsebene ist es nötig, eine Kommunikation zwischen den Anwendungen zuzulassen, bei der keine Annahmen über Klassen, Schnittstellen oder Funktionsdefinitionen anderer Anwendungen angestellt werden müssen. Weiterhin ist es wünschenswert, Module zur Erfüllung einer Aufgabe erst dann zu starten, wenn die Funktion tatsächlich benötigt wird, z.B. das Aufnehmen eines Bildes oder die Auswahl eines Kontakts. Die Kamera oder das Adressbuch soll nicht die meiste Zeit ungenutzt im Hintergrund Speicher belegen und Rechenzeit verbrauchen, sondern durch die Laufzeitumgebung erst dann aktiviert werden, wenn sie benötigt werden. Zu diesem Zweck richten die Anwendungen keine Listener ein – dazu müssten sie ja bereits laufen –, sondern die Kommunikation wird über sogenannte *Intents* abgewickelt. Mittels des *Intents* drückt eine Anwendung eine Absicht aus und überlässt es dem Laufzeitsystem, jemanden zu suchen, der diesen Zweck erfüllen kann. Die Nachricht, die das *Intent* repräsentiert, kann neben der eigentlichen Absicht auch viele weitere Informationen enthalten, die ein Empfänger zur Ausführung benötigt. So kann die Adressbuchanwendung nicht nur zur Auswahl von Kontakten dienen, sondern man kann auch neue Kontakte per Intent erstellen. Unsere Anwendung muss also nichts über die Implementierung des Adressbuchs wissen, sondern kann einfach seine Absicht ausdrücken, einen neuen Kontakt anlegen zu wollen.

Wie wir im Folgenden sehen werden, kommuniziert die gesamte Benutzeroberfläche – die selbst eine Anwendung ist – mit den anderen Anwendungen über Intents. Selbst das Starten einer Anwendung über das jeweilige Icon und das Eintragen der Applikation im sogenannten App-Launcher wird über Intents geregelt. Auch innerhalb einer Anwendung werden unterschiedliche Activities über Intents ausgelöst.

Wir müssen also bei der Programmierung mit zwei unterschiedlichen Modellen umgehen, die für unterschiedliche Nachrichtenflüsse eingesetzt werden:

1. den Intents
2. den Managern und Listenern

3.8 Intents (Absichten, Zwecke, Ereignisse)

Funktionen innerhalb der drei Kernanwendungskomponenten *Activities*, *Services* und *Broadcast-Receivers* werden über Nachrichten, sogenannte *Intents*, ausgelöst.

Ein *Intent* wird durch ein *Intent-Objekt* repräsentiert. Das *Intent-Objekt* beschreibt die Aktion, die ausgeführt werden soll, und transportiert ggf. weitere Informationen, die zur Erfüllung der Aufgabe benötigt werden.

Die Basisklasse für Intent-Objekte ist die Klasse `android.content.Intent`. Mit dieser Klasse konstruiert man Intent-Objekte, um Aktionen auszulösen, bzw. man erhält ein solches Objekt, wenn eine Aktion unserer Anwendung ausgelöst wurde.

Die Anwendungskomponenten definieren innerhalb des Manifests ihre *Intent-Filter*. Mittels der *Intent-Filter* legt eine Komponente fest, auf welche *Intents* sie unter welchen Bedingungen reagiert. Da diese Festlegung Bestandteil des Manifests ist, kann das Laufzeitsystem diese Informationen auswerten, ohne dass die Anwendung selbst läuft. Wenn eine Anwendung installiert wird vermerkt das Laufzeitsystem die entsprechenden *Intent-Filter* und die dazugehörige Anwendung. Wenn nun ein *Intent* ausgelöst wird, sucht das Laufzeitsystem über die *Intent-Filter* die entsprechende Anwendung bzw. Anwendungskomponente und bringt diese bei Bedarf zur Ausführung. Die Anwendungskomponente ist dann dafür verantwortlich, das *Intent-Objekt* auszuwerten und die entsprechende Funktion auszuführen.

Ein Intent wird über folgende Eigenschaften beschrieben:

Category	Kategorie der Anwendung. Die Kategorie legt die Art der Komponente fest, z.B. ob sie zum Application Launcher gehört und damit in die Liste der verfügbaren Anwendungen aufgenommen wird oder ob die Anwendung aus dem Browser durch Anklicken eines Links aufgerufen werden kann.
Action	Auszuführende Aktion, z.B. Starten der Hauptaktivität einer Anwendung, Initiieren eines Telefonanrufs oder Aufnehmen eines Bildes mit der Kamera.
Data	Lokalisierung von Daten, auf denen/mit denen die Aktion ausgeführt wird. Hier wird eine URI (Adresse) angegeben, über die die eigentlichen Daten lokalisiert und benutzt werden können. Solche Daten werden in der Regel von Content-Providern zur Verfügung gestellt, können aber auch irgendwo im Internet oder auf Speichermedien lokalisiert sein.

Tabelle 3.2: **Haupteigenschaften des Intent-Objekts**

Extras	Zusätzliche Informationen als Schlüssel-Wert-Paare, die die Zielkomponente auswerten und benutzen kann, z.B. die Telefonnummer beim Initiieren eines Anrufs.
Flags	Zusätzliche Informationen, die das Android-Laufzeitsystem auswertet, um zu wissen, wie die Komponente gestartet werden soll.

Tabelle 3.2: **Haupteigenschaften des Intent-Objekts (Forts.)**

Die Eigenschaft `Category` ist eine Zeichenkette. Das Android-System definiert in der Klasse `Intent` und in den unterschiedlichen Klassen des SDK Standardaktionen und Standardkategorien als Zeichenkettenkonstanten, z.B. die Aktion `Intent.ACTION_MAIN` mit dem Wert `android.intent.action.MAIN` oder die Kategorie `Intent.CATEGORY_LAUNCHER` mit dem Wert `android.intent.category.LAUNCHER`.

Wenn wir Intents konstruieren benutzen, wir immer die Konstanten der jeweiligen Klasse, bei der Definition von Intent-Filtern im Manifest wird die zugehörige Zeichenkette benutzt.

CODE

```
startActivity(new Intent(Intent.ACTION_EDIT, noteUri));
...
<activity android:name=".Spielwiese3">
<intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.categroy.LAUNCHER">
</intent-filter>
</activity>
```

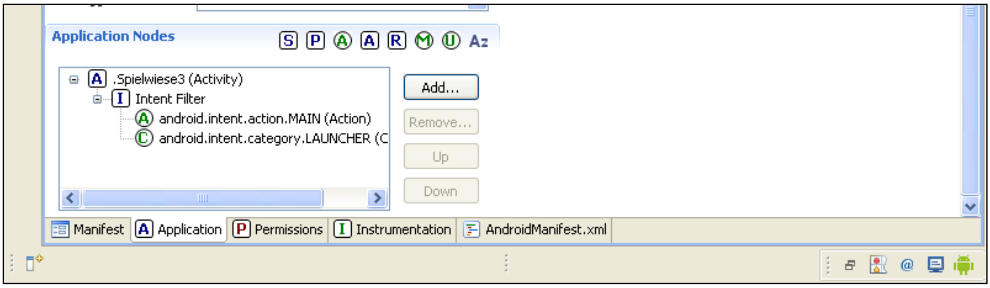


Abbildung 3.8: **Intent-Filter für die Activity Spielwiese 3 im Manifest-Editor**

Wenn wir für unsere Komponenten selbst Aktionen und/oder Kategorien definieren lehnen wir uns an die Benennung des SDK an. Die Konstanten werden in Großbuchstaben verfasst und mit dem Präfix `ACTION_` bzw. `CATEGORY_` versehen. Die Zeichenketten sollten immer aus dem voll qualifizierten Klassennamen in Kleinbuchstaben zuzüglich der Bezeichnung `category` bzw. `action` und dem Namen der Kategorie in Großbuchstaben aufgebaut sein:

CODE

```
package de.androidpraxis.Spielwiese;
    ...
    public class Spielwiese extends Activity {
    public static final ACTION_TUWAS = "de.androidpraxis.spielwiese.spiel➥
      wiese.action.TUWAS";
    ...
}
```

Was kann man nun mit den Intents anfangen?

1. Festlegen, auf welche Ereignisse unsere Anwendung wie reagiert
2. Andere Anwendungen ausführen, z.B. um eine Nachricht zu senden, ein Bild auszuwählen oder vieles mehr.

Der erste wichtige Intent-Filter, mit dem wir konfrontiert werden ist der Intent-Filter, um unsere Anwendung bzw. die Hauptaktivität der Anwendung im Application Launcher anzuzeigen und aufrufbar zu machen:

Listing 3.1: **Intent-Filter für die Activity Spielwiese3**

```
<activity android:name=".Spielwiese3">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.categroy.LAUNCHER">
    </intent-filter>
</activity>
```

Die Activity `Spielwiese3` bekommt als Intent-Filter die Action `ACTION_MAIN` (`android.intent.action.MAIN`) zugewiesen und wird in der Kategorie `CATEGORY_LAUNCHER` (`android.intent.categroy.LAUNCHER`) eingeordnet.

Der Application Launcher listet nun alle Activities auf, die diesen Intent-Filter deklariert haben, und erlaubt dem Anwender, diese Activity zu starten.

Wenn wir nun selbst andere Activities starten wollen, auch wenn es sich um Activities unseres eigenen Programms handelt, dann führen wir das auch so aus. Wir bestimmen über das Intent, was wir ausführen wollen, und starten die Activity.

Listing 3.2: **Starten einer Activity in der gleichen Anwendung**

```
class Spielwiese3 extends Activity
{
    […]
    public void starteSensorenActivity()
    {
        Intent mySensorsActivity = new Intent(this,Sensors.class);
        startActivity(mySensorsActivity);
    }
    […]
}
```

Listing 3.3: **Verschiedene Beispiele zum Aufruf anderer Aktivitäten**

```
class Spielwiese3 extends Activity
{
[…]
public void zeigeKontakt(Uri data) {
    Intent showContact = new Intent(Intent.ACTION_VIEW);
    showContact.setData(data);
    startActivityForResult(showContact,0);
}
```

```
public void waehleEinenKontakt()
{
    Intent picContact = new Intent(Intent.ACTION_PICK);
    picContact.setData(ContactsContract.Contacts.CONTENT_URI);
    startActivityForResult(picContact,MENU_PIC_CONTACT);
}

public void fotografiereEinBild()
{
    Intent picImage = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    startActivityForResult(picImage,0);
}

public void waehleEinBild()
{
    Intent picImage = new Intent(Intent.ACTION_GET_CONTENT);
    picImage.setType("image/*");
    startActivityForResult(picImage,0);
}
public void waehleEinBildAusContentProvider()
{
    Intent picImage = new Intent(Intent.ACTION_PICK);
    picImage.setData(MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
    startActivityForResult(picImage,0);
}
[…]
}
```

Im obigen Beispiel sind unterschiedliche Aufrufe zu sehen, die Vorgehensweise ist aber immer gleich. Wir initialisieren ein Intent-Objekt und setzen die benötigten Parameter. Die Action wird immer benötigt und bestimmt, was wir ausführen wollen. Zusätzlich geben wir mit `setData(…)` oder `setType(…)` an, auf welche Daten sich die Aktion bezieht. Ob man den Datentyp oder eine konkrete Adresse in Form einer URI angibt, hängt von der Aktion ab.

Die Typen und Adressen hängen wiederum sehr eng mit den Content-Providern zusammen, die als Baustein in Android dienen, um Daten gleich welcher Art zu verwalten, meist in einer SQLite-Datenbank, aber auch auf dem Dateisystem.

Als Resultat liefern die Activities oft wieder einen URI zurück, der die konkrete Adressierung der ausgewählten Daten beinhaltet.

Neben dem Zugriff auf die Daten, um z.B. den URI einer Adresse aus dem Adressbuch zu erhalten, können wir mittels der Intents auch Daten bearbeiten (ACTION_EDIT) oder auch, wenn der Content-Provider das erlaubt, Daten erstellen (ACTION_INSERT).

Wie wir sehen, sind die Intents der Dreh- und Angelpunkt zur Nutzung von auf dem Gerät installierten Diensten, wie der Adressbuchverwaltung, dem SMS und E-Mail-System oder des MediaStores, in dem Bilder und Videos verwaltet werden.

Durch das SDK sind bestimmte Aktionen vordefiniert, die bestimmte, immer wieder kehrende Aktionen standardisieren:

ACTION_MAIN	Eine Haupt-Activity der Anwendung, in Verbindung mit CATEGORY_LAUNCHER werden diese Activities in der Liste des Launchers angezeigt.
ACTION_VIEW	Anzeigen von Daten. Als Eingabe muss mittels Intent.setData(<Content-URI>) die Adresse der Daten/des Datums übergeben werden die/das wir anzeigen wollen. Wenn wir unsere Activity als Viewer für bestimmte Daten registrieren, müssen wir das im Intent-Filter mit dem <data ..>-Tag realisieren, mittels Intent.getData() erhalten wir den Content-URI der Daten, die angezeigt werden sollen.
ACTION_ATTACH_DATA	Zeigt an, dass Daten irgendwo anders angehängt werden sollen, tritt in der Regel als »Festlegen als...«-Menüpunkt auf, z.B. in der Bildgalerie. Wenn wir unsere Anwendung als Ziel für Bilddaten festlegen wollen, definieren wir einen Intent-Filter mit dieser Aktion und dem Mime-Type image/*. Mittels Intent.getData() können wir dann in unserer Activity den Content-URI des Bildes erfahren.
ACTION_EDIT	Ähnlich wie ACTION_VIEW, die übergebenen Daten sollen aber bearbeitet werden, z.B. der Dialog zum Bearbeiten eines Kontakts geöffnet werden.
ACTION_PICK	Auswählen eines Eintrags eines bestimmten Datentyps. In Intent.setData(...) wird der Content-URI (z.B: MediaStore.Images.Media.EXTERNAL_CONTENT_URI oder ContactsContract.Contacts.CONTENT_URI) angegeben und wir erhalten den ausgewählten Datensatz ebenfalls als Content-URI zurück.
ACTION_CHOOSER	Erlaubt die Auswahl aus alternativen Activities, die ein Intent bedienen können.
ACTION_GET_CONTENT	Ähnlich wie ACTION_PICK, der Unterschied ist dass wir hier den Typ angeben und nicht den URI: Intent.setType("image/*").
ACTION_DIAL	Startet die Anwendung zum Wählen einer Rufnummer. Die Rufnummer kann in setData(...) übergeben werden, entweder als Content-URI eines Kontakts oder in der Form »tel:08154711«. Der Anruf wird **nicht** direkt ausgeführt.
ACTION_CALL	Wie ACTION_DIAL, aber der Anruf wird direkt initiiert wenn eine Nummer oder ein Kontakt angegeben ist. Notrufnummern können damit **nicht** abgesetzt werden. Das geht mit ACTION_DIAL, der Anwender muss den Anruf dann noch aktiv bestätigen.

Tabelle 3.3: **Standardaktionen, um Activities zu starten**

ACTION_SEND	Daten senden. Als Parameter in Intent.put*Extra(...) stehen EXTRA_TEXT oder EXTRA_STREAM für die zu sendenden Daten zur Verfügung, und EXTRA_E-MAIL, EXTRA_BCC, EXTRA_CC und EXTRA_SUBJECT, um Empfänger und Betreff vorzubelegen. Mittels Intent.setType() muss der Mime Type der Daten gesetzt werden ("text/plain" bei EXTRA_TEXT), um generische Daten zu versenden, kann man "*/*" angeben. **Tipp:** ACTION_SEND kann man auch dazu verwenden, Daten an eine Anwendung zu senden. Das muss nicht unbedingt das tatsächliche Verschicken der Daten zur Folge haben. Damit können wir beliebige Anwendungen als Datenempfänger implementieren.
ACTION_SENDTO	Daten an jemanden senden, der Adressat wird mit Intent.setData(...) in Form eines Content-URI angegeben.
ACTION_ANSWER	Die Activity mit diesem Intent-Filter kann auf einen eingehenden Anruf reagieren.
ACTION_INSERT	Erstellen eines neuen Eintrags in einem Content-Provider. Mit intent.setData(...) wird der Content-URI des Content-Providers angegeben. Liefert den Content-URI des leeren Eintrags zurück (um z.B. dann ACTION_EDIT darauf aufzurufen).
ACTION_DELETE	Löschen eines Eintrags, in Intent.setData(...) wird der Content-URI des Eintrags angegeben.

Tabelle 3.3: **Standardaktionen, um Activities zu starten (Forts.)**

ACTION_TIME_TICK	Wird alle Minute gesendet. Man kann Empfänger nur innerhalb der Anwendung erstellen, es werden keine Empfänger über die Manifeste gesucht.
ACTION_TIME_CHANGED	Die Uhrzeit wurde neu eingestellt.
ACTION_TIMEZONE_CHANGED	Die Zeitzone hat sich geändert.
ACTION_BATTERY_CHANGED	Ladezustand der Batterie hat sich geändert. Kann nur innerhalb einer Anwendung empfangen werden, nicht über eine Deklaration im Manifest.
ACTION_BATTERY_LOW	Batterie wird schwach.
ACTION_BATTERY_OK	Batterie ist wieder o.k.
ACTION_POWER_CONNECTED	Externe Stromversorgung wurde entfernt.

Tabelle 3.4: **Broadcast Actions: Aktionen, auf die Broadcast Receiver reagieren können**

ACTION_POWER_DISCONNECTED	Externe Stromversorgung wurde angeschlossen.
ACTION_DEVICE_STORAGE_LOW	Speicher ist ziemlich voll.
ACTION_DEVICE_STORAGE_OK	Speicherfüllung ist nicht mehr kritisch.
ACTION_AIRPLANE_MODE_CHANGED	Das Gerät wurde in den Flugzeugmodus versetzt, dadurch werden manche Untersysteme (vor allem die mit Funk arbeiten wie WLAN, GSM) abgeschaltet, oder das Gerät wurde aus dem Flugmodus geweckt.
ACTION_MEDIA_UNMOUNTABLE	Externes Speichermedium kann nicht entfernt werden.
ACTION_MEDIA_UNMOUNTED	Externes Speichermedium wurde entfernt.
ACTION_MEDIA_MOUNTED	Externes Speichermedium wurde angeschlossen.
ACTION_SCREEN_ON	Bildschirm wurde angeschaltet.
ACTION_SCREEN_OFF	Bildschirm wurde abgeschaltet.
ACTION_USER_PRESENT	Der Benutzer ist präsent und hat das Gerät entriegelt.
ACTION_GTALK_SERVICE_CONNECTED	Das Gerät hat sich mit dem Google-Talk Dienst verbunden.
ACTION_GTALK_SERVICE_DISCONNECTED	Die GTalk-Verbindung wurde beendet.
ACTION_INPUT_METHOD_CHANGED	Die Eingabemethode wurde geändert.
ACTION_HEADSET_PLUG	Ein Headset wurde angeschlossen.
ACTION_NEW_OUTGOING_CALL	Ein ausgehender Anruf wurde platziert.
ACTION_CONFIGURATION_CHANGED	Die Gerätekonfiguration hat sich geändert (Ausrichtung, Sprache/Land ...).

Tabelle 3.4: **Broadcast Actions: Aktionen, auf die Broadcast Receiver reagieren können (Forts.)**

Viele der Broadcast-Aktionen signalisieren Änderungen am Zustand des Geräts. Diese Änderungen, besonders die Ausrichtung etc., werden vom System selbsttätig behandelt. Für unsere Anwendung kann es jedoch manchmal interessant sein, noch explizit auf bestimmte Änderungen reagieren zu können.

CATEGORY_DEFAULT	Standardaktion
CATEGORY_BROWSABLE	Die Aktion kann durch das Betätigen eines Links im Browser oder in WebViews für den Link-Typ ausgeführt werden.
CATEGORY_TAB	Die hinter der Aktion liegende Activity ist Bestandteil eines Tab-Hosts (Reiter).
CATEGORY_ALTERNATIVE	Die Aktion ist eine alternative Aktion für Datensätze. Es hängt sehr stark von den Anwendungen ab ob sie in ihr Optionenmenü alternative Aktionen aufnehmen.
CATEGORY_SELECTED_ALTERNATIVE	Die Aktion ist eine alternative Aktion für einen ausgewählten Datensatz. Es hängt sehr stark von den Anwendungen ab, ob sie in ihr Kontextmenü alternative Aktionen aufnehmen.
CATEGORY_LAUNCHER	Die Activity kann aus dem Startbildschirm oder dem Application-Launcher heraus gestartet werden. Außerdem werden diese Activities in der App-Liste angezeigt.
CATEGORY_HOME	Diese Activity ist die Home-Activity, d.h. die erste Activity, die nach dem Booten angezeigt wird. Könnte potenziell dazu dienen, einen eigenen Homescreen bereitzustellen.
CATEGORY_PREFERENCE	Die Activity dient für Einstellungen (PreferencePanel).
CATEGORY_CAR_DOCK	Activity wird ausgeführt, wenn das Gerät in eine Autohalterung eingesetzt wird.
CATEGORY_DESK_DOCK	Activity wird ausgeführt, wenn das Gerät in ein Dock eingesetzt wird.
CATEGORY_LE_DESK_DOCK	Activity wird ausgeführt, wenn das Gerät in ein Low-End Dock (analoges Dock) eingesetzt wird.
CATEGORY_HE_DESK_DOCK	Activity wird ausgeführt, wenn das Gerät in ein High-End Dock (digitales Dock) eingesetzt wird.
CATEGORY_CAR_MODE	Zeigt an, dass die Activity in einem Kfz-Umfeld genutzt werden soll/kann.
CATEGORY_APP_MARKET	Die Activity kann Applikationen auflisten und herunterladen. Damit könnte man seinen eigenen App-Store ansprechen.

Tabelle 3.5: **Kategorien, um Aktionen näher zu beschreiben**

Einige Standardanwendungen und Content-Provider definieren noch ihre eigenen Aktionen und Parameter. Der `MediaStore` z.B. implementiert unter anderem die Aktionen `ACTION_IMAGE_CAPTURE` und `ACTION_VIDEO_CAPTURE`, mit denen wir eine Anwendung zum Aufnehmen eines Bildes oder Videos starten können. Welche Aktionen verfügbar sind und wie diese parametrisiert werden kann man sich aus den jeweiligen Klassendokumentationen heraussuchen.

Welche Anwendung auf unsere Anfrage reagiert, braucht uns im Grunde nicht zu interessieren. Wir müssen nur darauf achten, dass wir die Ausnahme abfangen, wenn es gar keine Activity oder irgendeinen anderen Baustein gibt, der auf unsere Anfrage antworten könnte.

ACHTUNG

Viele Absturzursachen haben ihren Grund darin, dass von der Verfügbarkeit bestimmter Dinge ausgegangen wird. In einem lose gekoppelten System kann das aber zu optimistisch gedacht sein. Deshalb müssen wir darauf achten: Entweder finden wir vor dem Aufruf heraus, ob jemand auf unsere Anfrage reagieren könnte, oder wir fangen den Ausnahmefehler des Laufzeitsystems ab, um unsere Anwendung nicht abstürzen zu lassen.

Um herauszufinden, ob es zu einem Intent eine Komponente gibt, können wir den Package-Manager abfragen:

Listing 3.4: **Abfragen, ob zu einem Intent eine Komponente existiert**

```
public static boolean isIntentAvailable(Context context, final Intent in➥
    tent) {
    final PackageManager packageManager = context.getPackageManager();
    List<ResolveInfo> list =
    packageManager.queryIntentActivities(intent,
    PackageManager.MATCH_DEFAULT_ONLY);
    return list.size() > 0;
}
```

Nun kann es vorkommen, dass auf dem Gerät mehrere Anwendungen installiert sind, die eine Aufgabe durchführen können. Wenn wir z.B. etwas verschicken wollen (`ACTION_SEND`), dann kann das ja entweder per Mail, per SMS oder auch über anderweitige Transporte geschehen. Kann z.B. anhand der mitgegebenen Daten die entsprechende Anwendung nicht eindeutig identifiziert werden, wird der Application Chooser gestartet, aus dem wir dann die jeweilige Anwendung auswählen können.

Listing 3.5: **Beispiel »Senden einer Nachricht«**

```
public void sendeEineNachricht(String message, boolean withChooser)
{
    Intent theIntent = null;
    Intent sendMsg = new Intent(Intent.ACTION_SEND);
    sendMsg.putExtra(Intent.EXTRA_TEXT,message);
    sendMsg.setType("text/plain");
    if (withChooser)
    {
```

```
        theIntent = Intent.createChooser(sendMsg,getResources().getText(R.
string.anwendungZumSendenAuswahlen));
    }
    else
    {
        theIntent = sendMsg;
    }
    startActivityForResult(theIntent,0);
}
```

In diesem Beispiel wird per `ACTION_SEND` eine Textnachricht verschickt. Das kann nun entweder über das Mailprogramm, SMS, Bluetooth – oder was auch immer zur Verfügung steht – passieren.

Wenn wir selbst nicht `Intent.createChooser(…)` benutzen, dann bietet das Betriebssystem selbst einen Chooser an.

ACHTUNG

Der Chooser des Betriebssystems sagt aber nur, dass man eine Anwendung für eine Aktion auswählen soll. Wenn wir mit `Intent.createChooser(…)` arbeiten, können wir den Titel selbst festlegen. Das ist auch der Grund, warum wir selbst einen Chooser anbieten sollten, in dem genauer erklärt wird, was gerade passiert.

INFO

Ich habe alternative Browser auf dem ICONIA installiert. Der Effekt war, dass nun die Klicks auf die Links den Application Chooser geöffnet haben mit der Aufforderung, eine Anwendung für die Aktion auszuwählen. Das war sehr verwirrend und sicher auch nicht im Sinne des Erfinders, denn der Browser hat nicht gesagt, warum ich mich entscheiden muss. Man kann zwar die Standardaktion dann auf den Browser seiner Wahl festlegen, aber das ist nicht sehr transparent. Deswegen präferiere ich, den Chooser selbst zu starten oder, wie im Falle des Browsers, mich selbst als vorzuziehende Aktion zu betrachten, wenn Aktionen aus meiner Anwendung heraus angefordert werden, die ich selbst behandeln kann.



Abbildung 3.9: **Eigene Meldung mit Intent.createChooser(…)**

Hier weiß man genau: Ich muss aussuchen, was ich zum Senden der Nachricht benutzen will. Hier könnte man die Überschrift natürlich noch deutlicher formulieren.

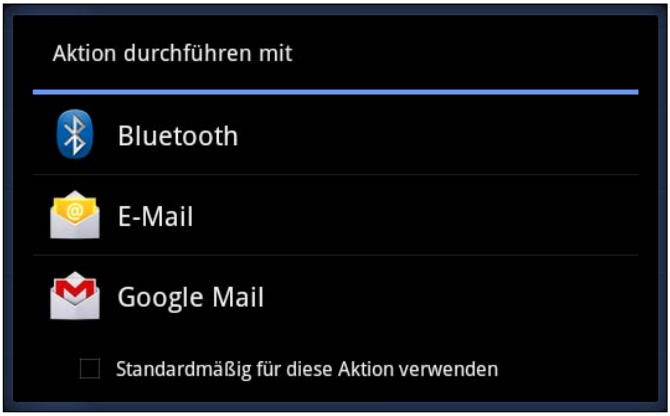


Abbildung 3.10: **Standard-Chooser des Betriebssystems**

Hier wissen wir nur, dass eine Aktion ausgeführt werden soll, aber nicht welche. Das ist manchmal vielleicht nicht so vertrauenserweckend, gerade wenn es ums Verschicken von irgendetwas geht.

Nun haben wir gelernt, wie wir mittels Intents mit anderen Anwendungen kommunizieren können, um z.B. ein Bild aus der Galerie oder einen Kontakt aus dem Adressbuch auszuwählen. Wie wir die Rückgabewerte verarbeiten, werden wir uns später noch anschauen. Außerdem werden wir noch weitere Dienste kennenlernen und uns auch über die Berechtigungen, die man für bestimmte Aktionen benötigt, Gedanken machen.

Wir können Funktionen unsere eigenen Applikation ebenfalls anderen Anwendungen über die Intent-Filter öffnen. Mittels des Intent-Filters ACTION_MAIN in der Kategorie LAUNCHER haben wir unsere Activity Spielwiese3 schon für den Zugriff aus dem Application Launcher geöffnet.

Um unsere Anwendung auch als Viewer für unsere Daten zu registrieren, legen wir einen entsprechenden Intent-Filter im Manifest an.

Listing 3.6: **Unsere Anwendung offeriert ACTION_VIEW für Ihre Spielwiesendaten.**

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="de.androidpraxis.Spielwiese3"
android:versionCode="1"
android:versionName="1.0">
<uses-sdk android:minSdkVersion="11" />

<application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".Spielwiese3"
    android:label="@string/app_name">
        <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```

```
    <activity android:name="SpielwiesenDatenViewer" android:label="@string/
datenviewer">
        <intent-filter>
        <action android:name="ACTION_VIEW"></action>
        <data android:mimeType="vnd.android.cursor.dir/x-androidpraxis-
spielwiesendaten"></data>
        <category android:name="android.intent.category.DEFAULT"></category>
        </intent-filter>
    </activity>
</application>
</manifest>
```

Eine weitere Anwendung für die Intent-Filter bzw. das Intent-Objekt ist, dass wir herausfinden können, ob es eine Activity für ein bestimmtes Intent gibt, und wir können uns auch alle Activities zu einem Intent auflisten lassen.

Ersteres ist nützlich, um z.B. Funktionen nur dann zu aktivieren, wenn ein Intent auch wirklich beantwortet werden kann:

Listing 3.7: **Prüfen, ob ein Intent verfügbar ist**

```
public class Utils {

    public static boolean isIntentAvailable(Context context, String action)
    {
        final PackageManager packageManager = context.getPackageManager();
        final Intent intent = new Intent(action);
        List<ResolveInfo> list =packageManager.queryIntentActivities(intent,
        PackageManager.MATCH_DEFAULT_ONLY);
        return list.size() > 0;
    }
}
[…]
if (Utils.isIntentAvailable(getApplicationContext(),ACTION_SEND)
{
[…]
}
```

Wenn wir uns alle Activities zu einem Intent auflisten lassen, z.B. für die Aktion MAIN in der Kategorie LAUNCHER, dann können wir selbst Dialoge oder Startbildschirme zum Start von Anwendungen schreiben. Hier schließt sich wieder der Kreis zur Offenheit und Modularisierung des Android-Systems. Durch die konsequente Nutzung des Intent-Systems ist es möglich, dass wir alternative Implementierungen für bestimmte Aufgaben zur Verfügung stellen, bzw. vorhandene Funktionen anderer Apps nutzen können.

3.9 Bausteine von Android-Applikationen

Die Applikationskomponenten (Application Components) sind die essenziellen Bausteine einer jeder Anwendung.

Manche der Bausteine sind der Einstieg für den Benutzer in unsere Anwendung und bilden die Benutzeroberfläche mit Menüs, Eingabeformularen, Listen, Animationen, manche Bau-

steine stellen einfach bestimmte Funktionen bereit, ohne eine eigene Benutzeroberfläche anzubieten und die nicht direkt durch den Benutzer aktiviert werden. Jeder Baustein ist für sich genommen eine eigene Einheit, der ein wohl definiertes Verhalten an den Tag legt.

Die Gesamtheit der Bausteine definiert das Verhalten und Aussehen unserer Anwendung.

Es gibt vier essenzielle Bausteine, aus denen wir unsere Anwendung bauen:

BAUSTEIN	WICHTIGE BASISKLASSEN
Activities und Fragments	Activity ListActivity ExpandableListActivity TabActivity PreferencesActivity Fragment ListFragment DialogFragment PreferenceFragment WebViewFragment
Services	IntentService Service
Content-Provider	ContentProvider
Broadcast Receiver	BroadcastReceiver AppWidgetProvider

Tabelle 3.6: **Übersicht über die Anwendungsbausteine**

Diese vier Bausteine sind die Bausteine, aus denen wir unsere Applikationen zusammenbauen. Wir entwickeln die Oberfläche und Funktionalitäten in den Activities, stellen Dienste (Services) bereit, die etwas im Hintergrund tun, verwalten Daten in einer Datenbank und stellen diese Daten mittels Content-Providern anderen Bausteinen zur Verfügung und reagieren mit Broadcast Receivern auf systemweite Ereignisse und Nachrichten. Die Funktion der Bausteine füllen wir selbst.

Androids Laufzeitumgebung bietet weitere Bausteine an, die wir in unserer Applikation selbst nutzen können. Entweder sind diese Bausteine selbst wieder Anwendungen, die auf einer Ebene mit unserer Anwendung laufen, oder es sind Bausteine der Laufzeitumgebung.

Auf die Bausteine der Laufzeitumgebung greifen wir über den Kontext der Applikation zu (Application Context) und fordern damit Schnittstellen zu Bausteinen der Laufzeitumgebung an oder führen Funktionen innerhalb der Laufzeitumgebung aus.

Der Kontext ist damit das Bindeglied zwischen der Laufzeitumgebung und unserer Applikation.

Folgende Services stellt das Laufzeitsystem bereit:

NAMENSKONSTANTE	SCHNITTSTELLE
WINDOW_SERVICE	WindowManager
LAYOUT_INFLATER_SERVICE	LayoutInflater
ACTIVITY_SERVICE	ActivityManager
POWER_SERVICE	PowerManager
ALARM_SERVICE	AlarmManager
NOTIFICATION_SERVICE	NotificationManager
KEYGUARD_SERVICE	KeyguardManager
LOCATION_SERVICE	LocationManager
SEARCH_SERVICE	SearchManager
SENSOR_SERVICE	SensorManager
STORAGE_SERVICE	StorageManager
VIBRATOR_SERVICE	Vibrator
CONNECTIVITY_SERVICE	ConnectivityManager
WIFI_SERVICE	WifiManager
AUDIO_SERVICE	AudioManager
TELEPHONY_SERVICE	TelephonyManager
INPUT_METHOD_SERVICE	InputMethodManager
UI_MODE_SERVICE	UiModeManager
DOWNLOAD_SERVICE	DownloadManager

Tabelle 3.7: **Übersicht über die Services der Laufzeitumgebung**

Ein weiterer essenzieller Baustein sind die Application Resources. In den Application Resources verwalten wir so ziemlich alles an (vordefinierten) Daten, die wir innerhalb unserer Applikationsbausteine verwenden wollen, und hauptsächlich sind das Daten, die in irgendeiner Form unsere Benutzeroberfläche ausmachen, also Layouts, Bilder, Texte, Menüs, verschiedene Stile etc.

Es ist wichtig, die Ressourcen intensiv zu nutzen und wo immer es geht auf die statische Programmierung von Texten oder Layouts zu verzichten. Das Ressourcensystem der Android-Laufzeitumgebung stellt mächtige Mechanismen zur Verfügung, unterschiedliche gerätespezifische Ressourcen vorzuhalten, z.B. Texte für unterschiedliche Sprachen, Layouts für unterschiedliche Bildschirmauflösungen und Bildschirmausrichtungen etc. Die Schnitt-

stelle zum Ressourcensystem nimmt uns die Arbeit ab, die richtigen Ressourcen herauszusuchen. Das erledigt das Ressourcensystem abhängig von unserer Gerätekonfiguration (Sprache, Bildschirm, etc.) vollkommen automatisch.

Im Folgenden wollen wir uns die Bausteine etwas genauer ansehen.

3.10 Application Resources

Die Application Resources werden unterhalb des `res`-Verzeichnisses unserer Android-Projekte verwaltet.

3.10.1 Grundlegende Struktur

Folgende Ressourcentypen werden bereitgestellt:

WAS	VERZEICHNIS/ DATEI	BESCHREIBUNG
Assets	assets	Unterschiedliche, untypisierte Dateien, die mittels AssetManager gelesen werden können.
Animation Resources	res/anim	Definition von Animationen, die auf View-Elemente oder für allgemeine Objektanimationen angewendet werden kann.
Property Animation Resources	res/animator	Animationsressourcen für das Property Animation Framework. Diese Animationsressourcen sind nicht auf Views beschränkt, sondern können zur Animation beliebiger Objekte benutzt werden.
Color State Lists	res/color	Definieren von Farblisten für die verschiedenen Zustände eines View-Elements (Normal, Gedrückt, Fokussiert etc.). Mit Farblisten kann z.B. ein Button mit unterschiedlichen Farben belegt werden, je nach dem ob er gerade gedrückt ist oder nicht.
Drawables	res/drawable	Hier können alle möglichen Objekte definiert werden, die irgendwie auf dem Bildschirm dargestellt werden können. Dazu gehören z.B. Bitmaps (Bilder, Symbole), geometrische Figuren, Hintergründe für verschiedene View-Status etc.

Tabelle 3.8: **Übersicht über die verschiedenen Ressourcentypen**

Layouts	res/layout	In den Layouts werden die Oberflächen für Activities oder eigene Benutzerinterface-Komponenten definiert. Neben den Drawables, Strings und Menue-Ressourcen ist das die erste wichtige Ressource, die wir benutzen.
Menues	res/menu	Definition von Menüstrukturen für Optionsmenüs oder Kontextmenüs.
Raw	res/raw	Untypisierte Daten, auf die mittels des ResourceManagers geladen werden kann. Der Zugriff erfolgt über eine eindeutige ID, im Gegensatz zu Assets, auf die mittels Dateinamen zugegriffen wird.
Strings	res/values/ strings.xml	In den Strings können Zeichenketten hinterlegt werden, auf die über einen Bezeichner/Namen zugegriffen werden kann. Wenn wir in den Anwendungen Texte verwenden, egal ob in anderen Ressourcen wie den Layouts oder direkt im Programm, sollten wir die Texte immer über die Strings referenzieren. Das macht die Internationalisierung leichter, und auch Textänderungen und Wiederverwendung werden vereinfacht.
Styles	res/values/ styles.xml	Hier können wir über Stylesheets (Stilvorlagen) das Aussehen unserer Elemente der Benutzeroberflächen bestimmen. Anstatt z.B. Schriftgrößen und Schriftarten direkt bei den Elementen zu definieren können wir das über Stilvorlagen erledigen. Stile können zu sog. Themes (Themen) zusammengefasst werden. Damit lässt sich dann das Aussehen einer ganzen Activity entsprechend anpassen und ggf. auch zur Laufzeit wechseln.
Weitere einfache Ressourcentypen: Bool Color Dimension ID Integer IntegerArray TypedArray	z.B. res/values/ arrays.xml res/values/ cloros.xml res/values/ dimens.xml res/values/ id.xml	Hier können verschiedene Ressourcentypen als Name-Wert-Paar definiert werden, z.B. Farben, Dimensionen etc., auf die dann über den Namen zugegriffen werden kann. Ein großer Vorteil ist, dass bei einigen dieser Typen wie z.B. der Dimensionstypen beim Zugriff eine Umwandlung des Werts abhängig von der Gerätekonfiguration stattfindet. Damit kann man sich die eigene Umwandlung innerhalb der Anwendung sparen.

Tabelle 3.8: **Übersicht über die verschiedenen Ressourcentypen (Forts.)**

Die Ressourcen selbst werden immer in Dateien abgelegt, oft in XML-Dateien, andere Dateien wie Bilder natürlich in ihrem jeweiligen Format (z.B. .png, .jpg).

Bei einfachen Ressourcentypen, die unter dem Verzeichnis res/values organisiert werden, spielt der Name der XML-Datei im Grunde keine Rolle. Es hat sich allerdings eingebürgert dass die XML-Datei wie der Datentyp benannt wird, siehe die Beispiele in obiger Tabelle. Die Konstante zum Zugriff auf die Werte wird aus dem `name`-Attribut der einfachen Ressource gebildet.

Bei den komplexeren Ressourcen wird der XML-Dateiname zu einer Konstante innerhalb unserer Anwendung, auf die per *`R.<Resourcetype>.<Name ohne XML>`* zugegriffen werden kann. Ein Layout wird z.B. in einer Activity mittels *`setContentView(R.layout.meinlayout)`* benutzt. Im `res/layout` Verzeichnis gibt es eine entsprechende Datei `meinlayout.xml`. Genauso werden auf Drawables, z.B. Bitmaps, über den zu einer Konstante umgewandelten Dateinamen zugegriffen.

ACHTUNG

Per Definition werden diese Konstanten alle in Kleinbuchstaben verwendet. Die entsprechenden Dateien müssen auch komplett kleingeschrieben werden und dürfen keine Sonderzeichen und keine Umlaute enthalten, und sie dürfen nicht mit einer Zahl beginnen. Die Dateien müssen so benannt werden, dass sich ein gültiger Java-Konstantenname ergibt.

Die Dateien müssen unterhalb der jeweiligen Verzeichnisse in `res/<resourcetyp>` abgelegt werden, es dürfen keine Dateien in *`res`* direkt abgelegt werden.

Die Verzeichnisse der obigen Aufzählung sind die Standardverzeichnisse der Ressourcen. Ressourcen, die dort abgelegt werden, sind die Standardressourcen und werden geladen, wenn keine alternativen, gerätespezifischen Ressourcen vorliegen.

Es ist wichtig, diese sogenannten *Default Resources* bereitzustellen, auch wenn man gerätespezifische, konfigurationsabhängige Ressourcen verwendet. Die Default Resources werden immer benutzt, wenn keine gerätespezifische Ressource gefunden wurde. Liefert man keine Default Resource mit und eine gerätespezifische Ressource wurde nicht gefunden, dann läuft die Anwendung auf einen Fehler.

Für einige Standardressourcen, speziell die Drawables, führt das Ressourcensystem beim Laden möglicherweise Transformationen durch, um z.B. die Ausmaße eines Bildes oder einer geometrischen Figur auf unterschiedlichen Auflösungen anzupassen. Die Bezugssauflösung der Default Resources ist immer 160 dpi (*mdpi*). Wird eine solche Default Resource auf einem *hdpi*-Gerät geladen, wird die Ressource entsprechend hochskaliert, natürlich mit dem Nachteil, dass sich ggf. Aliasartefakte durch das Hochskalieren ergeben, das Bild z.B. Klötzchen bildet.

3.10.2 Konfigurationsabhängige alternative Ressourcen

Jede Anwendung sollte also alternative Ressourcen anbieten, um gerätespezifische Konfigurationen wie unterschiedliche Auflösungen und Abmessungen zu unterstützen.

Auch die Unterstützung verschiedener Sprachen wird durch alternative Ressourcen realisiert.

Gerätespezifische Ressourcen werden in Unterverzeichnissen unterhalb `res` abgelegt, die nach einem definierten Benennungsschema benannt sind. Der Name des Verzeichnisses beginnt mit dem Ressourcentyp (`res/drawable`) und wird um qualifizierte Konfigurationsnamen, die mit einem Bindestrich getrennt werden, ergänzt: `res/drawable-hdpi` für Ressourcen, die auf einem *hdpi*-Gerät benutzt werden sollen.

Allgemein werden die alternativen Ressourcen so gebildet:

```
res/<resourcetyp>[-<qualifiervalue>][-<qualifiervalue>][…]
```

ACHTUNG

Wir werden später sehen, dass zum Zugriff auf die Ressourcen die Ressourcen-ID benutzt wird, die aus dem Ressourcentyp und dem Namen der Ressource gebildet wird. Es gilt zu beachten dass wir immer auf eine Ressource eines bestimmten Typs zugreifen, aber niemals angeben können, dass wir eine Ressource eines bestimmten Klassifizierers laden möchten. Den richtigen Klassifizierer wählt das Ressourcensystem beim Laden der Ressource selbst aus.

Auf einem *hdpi*-Gerät würde das Ressourcensystem die entsprechenden Ressourcen aus `res/drawable-hdpi` laden.

Folgende Konfigurationstypen können benutzt werden:

MCC und MNC	Das sind der Mobile Country Code und der Mobile Network Code. Mit dem MCC wird das Land gekennzeichnet, in dessen Mobilfunknetz sich unser Gerät gerade befindet, der MNC kennzeichnet den Netzbetreiber.	mcc310 (U.S., any carrier) mcc310-mnc004 (U.S. on Verizone)
Sprache und Region	ISO-639-1 Sprachcode (zwei Buchstaben) gefolgt von einem ISO 3166-1-alpha-2 Regionencode, dem ein kleines r vorangestellt wird. Die Groß-/Kleinschreibung der Codes ist egal, das kleine r für die Kennzeichnung des Regionencodes muss aber tatsächlich klein sein. Eine Region kann nicht **ohne** einen Ländercode stehen. **Sprache und Region können sich dynamisch ändern.**	de en fr en-rUS fr-rFR fr-rCA

Tabelle 3.9: **Übersicht über die Konfigurationsqualifizierer für Ressourcen**

Bildschirmgröße	Der Qualifizierer orientiert sich an der Bildschirmdiagonalen (alles ca. Angaben): small: 2" bis 3,5" normal: 3" bis 4,5" large: 4,3" bis 7" xlarge: ab 7"	small normal large xlarge
Seitenverhältnis	Das Seitenverhältnis ist die Generalisierung des eigentlichen Seitenverhältnisses. long: WQVGA (5:3), WVGA (5:3), FWVGA (16:9) notlong: QVGA (4:3), HVGA (3:2), VGA (4:3)	long notlong
Bildschirm-ausrichtung	port: Hochformat land: Querformat **Die Ausrichtung kann sich dynamisch ändern**	port land
Dockingmodus	car: Das Gerät wurde im Auto in eine DockingStation eingesetzt. desk: Das Gerät wurde in eine DockingStation auf dem Schreibtisch eingesetzt. **Der Dockingmodus kann sich dynamisch ändern.**	car desk
Nachtmodus	Basiert auf der aktuellen Tageszeit. **Der Nachtmodus kann sich (selbstverständlich) dynamisch ändern.**	night notnight
Pixeldichte (dpi)	Bezieht sich auf die Punktdichte in der Bilddiagonalen. ldpi: Niedrige Dichte, ca. 120dpi mdpi: Mittlere Dichte, ca. 160dpi hdpi: Hohe Dichte, ca. 240dpi xhdpi: Sehr hohe Dichte, ca. 320dpi nodpi: Dieser Qualifizierer kann für Bitmaps genutzt werden, die **nicht** skaliert werden sollen.	ldpi mdpi hdpi xhdpi nodpi
Typ des Touch-screens	notouch: Kein Touchscreen stylus: Touchscreen der für Stiftbedienung geeignet ist finger: Touchscreen für Bedienung mit dem Finger	notouch stylus finger

Tabelle 3.9: **Übersicht über die Konfigurationsqualifizierer für Ressourcen**

Tastatur-verfügbarkeit	keysexposed: Eine Tastatur ist gerade geöffnet und bedienbar. Das kann sowohl eine echte als auch die virtuelle Tastatur sein. Selbst wenn eine echte Tastatur vorhanden, aber eingeklappt ist und die virtuelle Tastatur angezeigt wird, gilt dieser Qualifizierer. keyshidden: Es gibt eine echte Tastatur, diese ist aber eingeklappt/zugeklappt, und eine virtuelle Tastatur ist ebenfalls nicht aktiv. keyssoft: Es gibt eine virtuelle Tastatur. Die Sichtbarkeit ist nicht von Belang. **Die Tastaturverfügbarkeit kann sich dynamisch ändern.**	keysexposed keyshidden keyssoft
Primäre Eingabe-methode	Dieser Qualifizierer bezieht sich auf echte Tastaturen. Wenn das Gerät nur eine virtuelle Tastatur hat, dann gilt nokeys. nokeys: Das Gerät hat keine echte Tastatur. qwerty: Das Gerät hat eine Schreibmaschinentastatur. 12key: Das Gerät hat eine 12-Tasten-Telefontastatur.	nokeys qwerty 12key
Verfügbarkeit von Navigationstasten	navexposed: Navigationstasten sind sichtbar navhidden: Navigationstasten sind nicht sichtbar/eingeklappt/zugeklappt. **Die Verfügbarkeit von Navigationstasten kann sich dynamisch ändern.**	navexposed navhidden
Primäre Navigationsmethode (außer Touchscreen)	nonav: Es gibt keine Navigationstasten dpad: Ein D-Pad ist vorhanden (Das ist die Kreuzwippe, die man von Gameboy- oder ähnlichen Controllern kennt). trackball: Es steht ein Trackball zur Verfügung wheel: Es stehen Drehräder zur Verfügung. Diese Konfiguration ist nicht gebräuchlich.	nonav dpad trackball wheel
Version der Android-Plattform (API Level)	Bestimmt die Version der Plattform, auf der die Anwendung läuft.	v11 v10 v9 ...

Tabelle 3.9: **Übersicht über die Konfigurationsqualifizierer für Ressourcen (Forts.)**

Die Namen werden nach folgenden Regeln gebildet:

1. Es können mehrere Qualifizierer für einen Satz an Ressourcen benutzt werden, die Werte werden mit Bindestrichen aneinandergehängt.
2. Die Werte müssen in der Reihenfolge notiert werden, wie sie in der Tabelle aufgeführt sind.
3. Es darf immer nur ein Wert für einen Qualifizierer angegeben werden.

Um z.B. den Satz von grafischen Elementen für die Sprache Französisch auf hdpi-Geräten zu benennen, legt man folgendes Verzeichnis an: `res/drawable-fr-hdpi`.

Die Tabelle zeigt die Qualifizierer in ihrer logischen Reihenfolge. Landes- und sprachspezifische Varianten stehen ganz oben in der Hierarchie, da dies z.B. bei Texten, aber auch bei landes- und sprachspezifischen Grafiken das oberste Entscheidungskriterium ist. Die weniger wichtigen Konfigurationseigenschaften finden sich ganz unten. Eine Klassifizierung der Ressourcen nach verfügbaren Navigationstasten wird wohl relativ selten benötigt.

Die Reihenfolge muss auch eingehalten werden (Regel 2). Richtig ist: `res/drawable-fr-hdpi`, falsch ist: `res/drawable-hdpi-fr`.

Worauf man achten muss, ist, dass man nicht einen Ressourcensatz für mehrere Werte eines Qualifizierers benutzen kann (Regel 3). Wenn wir z.B. eine Flagge haben, die sowohl für das französische Kanada als auch das englischsprachige Kanada gültig ist, kann man Folgendes **nicht** machen: `res/drawable-en-rCA-fr-rCA/ca_flagge.png`.

Man sollte die Flagge natürlich auch nicht als Default anlegen, und eine Region kann auch nicht alleine ohne Landeskennzeichen qualifiziert werden. Es müssen also zwei Verzeichnisse erstellt werden:

`res/drawable-en-rCA/ca_flagge.png`

`res/drawable-fr-rCA/ca_flagge.png`

Um zu vermeiden dass man Dateien in so einem Fall duplizieren muss, kann man eine Alias-Ressource anlegen, die auf eine gemeinsam genutzte Datei verweist.

```
res/drawable/ca_flagge.png
res/drawable-en-rCA/ca_flagge.xml
res/drawable-fr-rCA/ca_flagge.xml
…
ca_flagge.xml:
<?xml version="1.0" encoding="utf-8"?>
<bitmap xmlns:android="http://schemas.android.com/apk/res/android"
    android:src="@drawable/ca_flagge" />
```

Alternative Ressourcen werden in der Regel für sprach- und landesabhängige Inhalte (Texte, Bilder) in Verbindung mit bildschirmabhängigen Inhalten (Layouts, Drawables) verwendet.

Layouts und Drawables hängen oft von der Bildschirmgröße und der Ausrichtung ab. Die Texte innerhalb des Layouts sind ja selbst Referenzen auf Strings, die Strings (und ggf. verwendete Drawables) selbst wiederum sind sprach- und landesabhängig. Daher kommt man in der Regel nicht in Versuchung, ein Layout sprachabhängig zu verwalten.

Mögliche Ressourcenkonstellationen sind z.B.:

res/values/strings.xml	Texte in der Standardsprache/Standardkonfiguration (z.B. alle Texte auf Deutsch, wenn unsere Applikation vornehmlich in deutscher Sprache daherkommt)
res/values-en/strings.xml	Die englische Übersetzung
res/values-fr/strings.xml	Die französische Übersetzung
res/drawable/...	Standard-Drawables, Skalierung auf Zielauflösung möglich, für alle Sprachen gültig
res/drawable-en/...	Drawables für Englisch, ggf. Skalierung auf Zielauflösung
res/drawable-hdpi/...	Drawables für hdpi-Auflösung. Skalierung auf höhere Auflösung möglich, sprachübergreifend
res/drawable-en-hdpi/...	Die englischen hdpi-Drawables
res/drawable-land/...	Drawables für Landscape-Ausrichtung (z.B. Hintergründe, die entsprechend der Ausrichtung gestaltet sind)
res/drawable-hdpi-land/...	Drawables für Landscape-Ausrichtung in hdpi-Auflösung (z.B. Hintergründe, die entsprechend der Ausrichtung gestaltet sind)

Tabelle 3.10: **Übersicht über sinnvolle Ressourcenkombinationen**

TIPP

Die konsequente Nutzung des Ressourcensystems sollte zur Pflicht werden. Selbst bei kleinen Programmen sollten wir nie Zeichenketten direkt verwenden, sondern immer in String-Ressourcen auslagern. Das gilt sowohl für Zeichenketten innerhalb des Programmcodes als auch für Zeichenketten in Layouts, z.B. Eingabeformularen. Durch nur wenig mehr Aufwand bereiten wir unsere Programme direkt für den Einsatz in mehreren Sprachen vor.

Auch das Auslagern von Werten wie den Abmessungen (Dimensions) wird auf lange Sicht neben dem etwas höheren Pflegeaufwand einen großen Nutzen bringen, denn man kann hier direkt die implizite Umwandlung auf die Bildschirmgeometrie nutzen, ohne diese Berechnung selbst ausführen zu müssen.

3.10.3 Ressourcen-IDs

IDs für Ressourcen werden durch das aapt-Tool (Android Asset Packaging Tool) automatisch generiert und als Java-Konstanten in der Klasse R abgelegt. Die Java-Klasse R dürfen wir nie per Hand verändern, da die Klasse immer neu durch das aapt-Tool erstellt wird.

Für die einzelnen Ressourcentypen legt das aapt-Tool innerhalb von R weitere Klassen an, die die einzelnen Konstanten beinhalten.

Wie die Konstante gebildet wird, hängt vom Ressourcentyp ab. Handelt es sich im einen einfachen Typ wie Strings oder Dimensions, wird der Name der Konstante aus dem android:name-Attribut gebildet:

```
[…]
    <string name="item_startgame">Starten</string>
[…]
```

ergibt:

```
[…]
public final class R {
    […]
    public static final class string {
        […]
        public static final int item_startgame=0x7f040002;
    }
}
```

Bei komplexen Ressourcentypen wie Layouts, Animationen, Menüs, Drawables, Color State Lists und Styles wird die Ressourcen-ID aus dem Dateinamen erzeugt:

```
res/layout/main.xml
res/menu/gamemenu.xml
```

ergibt:

```
[…]
public final class R {
[…]
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class menu {
        public static final int gamemenue=0x7f050000;
    }
}
```

ACHTUNG

Nochmals der Hinweis: Die Dateinamen immer in Kleinbuchstaben schreiben und so benennen, dass ein gültiger Java-Bezeichner erstellt werden kann. Sollten einmal Übersetzungsfehler in der Klasse R auftauchen, liegt das oft an einem falschen Dateinamen.

Innerhalb von Layout-Ressourcen werden Benutzeroberflächen beschrieben, die aus einer View-Hierarchie und unterschiedlichen View-Elementen bestehen. Innerhalb unserer Anwendung benötigen wir häufig den Zugriff auf View-Elemente, um z.B. Daten an eine Listendarstellung zu übergeben oder den Text aus einem Texteingabefeld auszulesen.

Auf Elemente in dieser Hierarchie wird über eine ID zugegriffen. Diese ID ist keine Ressourcen-ID, sondern eine Element-ID, die innerhalb der Ressource für ein Element deklariert wird. Allerdings müssen die verwendeten IDs ebenfalls in den Ressourcen als Ressourcentyp-ID mit Namen und Wert deklariert werden.

Um ein Element mit einer ID zu versehen, benutzen wir in den XML-Dateien das android:id-Attribut:

```
[…]
<TextView android:id="@+id/hello_textview"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_text" />
[…]
```

Hier wird der TextView die ID `hello_textview` zugewiesen. Später kann man dann auf die TextView mittels `findViewById(R.id.hello_textview)` zugreifen.

TIPP

Wichtig ist die Notation der ID an dieser Stelle. Zu beachten ist die Schreibweise `@+id`, die sich von der üblichen Schreibweise zum Zugriff auf eine Ressource wie `@string/hello_text` unterscheidet. Das +-Zeichen sorgt dafür, dass das aapt-Tool die ID hello_textview neu erstellt, falls sie noch nicht existiert, und mit einem eindeutigen Wert belegt. Das ist eine feine Sache, denn dann müssen wir uns nicht darum kümmern, die ganzen IDs »von Hand« aufzuschreiben.

3.10.4 Zugriff auf Ressourcen

Wir verwenden Ressourcen in zwei Situationen:

1. Aus einer anderen Ressource heraus
2. Aus unserem Programm heraus

Die häufigste Verwendung von Ressourcen innerhalb anderer Ressourcen ist die Deklaration von Layouts und Menüs. Es ist offensichtlich, dass wir hier sehr häufig auf Zeichenketten, Drawables und Stile zugreifen wollen. Allgemein notieren wir den Zugriff innerhalb eines Attributs auf eine andere Ressource wie folgt:

```
@[<package_name>:]<resource_type>/<resource_name>
```

`<package_name>` ist der Name des Packages, aus dem wir die Ressource referenzieren wollen. Meistens arbeiten wir mit Ressourcen aus unserem eigenen Package, dann können wir den Package-Namen weglassen. Manchmal müssen wir Ressourcen aus dem Android-System selbst oder aber aus Bibliotheken referenzieren, die wir mit unserem Package ausliefern. Dann brauchen wir den Package-Namen, um die Ressource eindeutig zu referenzieren.

`<resource_type>` ist der Typ der Ressource (string,layout,drawable etc.).

`<resource_name>` ist der Name der Ressource, wie beschrieben entweder aus dem Dateinamen gebildet (layout,drawable etc.) oder aus dem Namen des XML-Elements (string ...).

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:title="@string/item_startgame" android:id="@+id/item_
startgame"></item>
    <item android:title="@string/item_settings" android:id="@+id/item_set➥
    tings"></item>
</menu>
```

oder:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <de.androidpraxis.marblegame.PlaygroundView
      android:id="@+id/playground"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:background="@drawable/playground_background"/>

</FrameLayout>
```

Für den Zugriff aus unserem Java-Programm heraus gilt ein ähnlich einfaches Prinzip, um die Ressourcen-ID zu bilden, so dass wir eigentlich nie in die R-Klasse direkt reinschauen müssen, wir müssen lediglich unsere Projektstruktur ansehen.

Die allgemeine Form lautet:

`R.[<package_name>.]<resource_type>.<resource_name>`

ACHTUNG

Es sei nochmals darauf hingewiesen dass der Ressourcentyp tatsächlich nur den Ressourcentyp (`drawable`, `layout`, ...) beinhaltet und zum Zugriff auf eine Ressource keine Klassifizierer wie `-de` oder `-hdpi` verwendet werden.

Um z.B. das Layout für eine Activity zu erstellen, verwenden wir folgendes Konstrukt in der onCreate()-Methode:

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    […]
    setContentView(R.layout.main);
    playgroundView = (PlaygroundView)findViewById(R.id.playground);
    playgroundView.setBackgroundDrawable(R.drawable.playground_hintergrund);
    […]
}
```

Bei folgendem `res/layout/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <de.androidpraxis.marblegame.PlaygroundView
      android:id="@+id/playground"
      android:layout_width="match_parent"
      android:layout_height="match_parent"/>
</FrameLayout>
```

und `res/drawable/playground_hintergrund.png`.

Der Zugriff auf das Ressourcensystem von Android wird durch die Klasse `Ressources` (`android.content.res.Resources`) realisiert. In jedem Kontext (Activities, Services, Broadcast-Receiver und Content-Provider) können wir mittels `getResources()` Zugriff auf das Ressourcensystem erhalten.

Wie im obigen Beispiel anhand von `setContentView()` und `setBackgroundDrawable()` zu erkennen, bieten einige Klassen, vornehmlich die View-Klassen, bereits Methoden an, die eine Ressourcen-ID erwarten und den nötigen Zugriff auf das Ressourcensystem intern abhandeln. Zur Nutzung dieser Methoden benötigen wir keinen direkten Zugriff auf die `Resources`-Klasse.

Für den Zugriff auf einfache Ressourcentypen wie Strings, Dimensions, Colors etc. müssen wir über die Klasse `Resources` zugreifen. Gerade für die Nutzung von Zeichenketten, aber auch die Nutzung von Dimensionsangaben werden wir diesen Zugriff häufiger benötigen.

Um z.B. auf einen String zuzugreifen, benutzen wir die Methode:

```
getResources().getString(R.id.begruessung);
```

Der Zugriff auf komplexe Ressourcen wie Layouts und Menüs, die als XML-Dokument deklariert sind, ist etwas komplizierter, denn diese Ressourcen werden ja als XML-Dokumente gespeichert. Durch das aapt-Werkzeug werden diese XML-Dokumente bereits in eine interne optimierte Struktur umgewandelt, auf die dann zugegriffen wird.

Für Layouts wird dazu der `LayoutInflater` benutzt, für Menüs der `MenuInflater`.

Der `LayoutInflater` wird seltener direkt benutzt, denn Activities stellen die Methode `setContentView()` zur Verfügung die das automatisch erledigt. Der Zugriff auf `MenuInflater` ist häufiger anzutreffen, da dieser die einzige Möglichkeit ist Menüs aus den Ressourcen zu erzeugen, sei es als Optionenmenü oder als Kontextmenü.

Wie die Zugriffe im Einzelnen aussehen, schauen wir uns bei den einzelnen Ressourcentypen an.

Das Ressourcensystem ist auf dem Asset-Manager aufgebaut und stellt Methoden bereit, um auf die unterschiedlichen Ressourcentypen zuzugreifen. Neben den Ressourcen, die wir für unsere Anwendung definieren, liefert der Asset-Manager bzw. das Ressourcensystem auch den Zugriff auf implizit vorhandene Ressourcen:

1. Bildschirmmetrik (`DisplayMetrics`):

   ```
   getResources().getDisplayMetrics();
   ```

2. Die aktuelle Konfiguration (`Configuration`):

   ```
   getResources().getConfiguration();
   ```

Über `DisplayMetrics` erhält man z.B. Informationen über die Bildschirmgröße und die Punktdichte (physikalische und logische dpi-Anzahl, sowie die tatsächlichen dpi in x- und y-Richtung).

Über `Configuration` erhält man z.B. Informationen über die **aktuelle** Bildschirmausrichtung (Hochformat oder Querformat), ob eine Tastatur vorhanden ist, welche aktuelle Spracheinstellung gewählt wurde etc.

INFO

In der Regel müssen wir auf diese grundlegenden Informationen nicht zugreifen, da das Ressourcensystem ja mittels der Klassifizierer z.B. die Spracheinstellungen oder das aktuelle Bildschirmformat beim Zugriff auf die Ressourcen schon berücksichtigt. Selbstverständlich kann es aber irgendwann mal nützlich sein, die Informationen selbst auszuwerten, z.B. wenn man die Änderung der Bildschirmausrichtung selbst behandeln möchte.

3.10.5 Einfache Ressourcen

Einfache Ressourcen werden in einer oder mehrerer XML-Dateien unter res/values bzw. den alternativen Ressourcenverzeichnissen organisiert. Wie bereits angesprochen, hat sich eingebürgert, die unterschiedlichen einfachen Typen in jeweils eigenen XML-Dateien zu

organisieren, die nach dem Typ benannt sind. Das ist aber kein Muss, man kann jeden beliebigen Namen wählen.

INFO

Im Gegensatz zu den komplexen Ressourcentypen hat der Dateiname der XML-Datei keinen Einfluss auf die Erzeugung von Ressourcen-IDs. Wir sollten eine Dateistruktur wählen, mit der wir einen optimalen Überblick über unsere einfachen Ressourcen behalten können, z.B. eine Benennung nach Moduln oder Ähnlichem. So könnten wir für das Layout `res/layout/kontakt_eingabe.xml` **auch ein korrespondierendes** `res/values/kontakt_eingabe_strings.xml` **anlegen, um die Zeichenketten dieses Benutzerinterfaces zu verwalten.**

Die Grundstruktur der jeweiligen XML-Datei sieht folgendermaßen aus:

Listing 3.8: **Grundstruktur der Ressourcendateien einfacher Ressourcen**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
[…]
</resources>
```

Die jeweiligen Typen werden wie folgt deklariert:

Bool

```
<bool name="…">[true | false]</bool>
<bool name="tolle_anwendung">true</bool>
boolean wert = getResources().getBoolean(R.id.tolle_anwendung);
```

Color

```
<color name="…">[#RGB | #ARGB | #RRGGBB | #AARRGGBB]<color>
```

Die Werte werden in hexadezimaler Notation aufgeführt und bestimmen den Rot- (R), Grün- (G) und Blau- (B)Wert der Farbe. Der Alpha-(A)Wert bestimmt die Transparenz der Farbe von 0 bzw. 00 (vollkommen transparent) bis F bzw. FF vollkommen deckend).

```
<color name="blauer_hintergrund">#0000ff</color>
int color = getResources().getColor(R.id.blauer_hintergrund);
```

TIPP

Ressourcen vom Typ Color können auch als Drawables referenziert werden. Das heißt, überall dort, wo wir z.B. den Hintergrund einer View auf eine bestimmte Hintergrundfarbe setzen wollen, können wir anstatt auf ein Drawable auch auf eine Farbe verweisen: `android:background="@color/blauer_hintergrund"`**.**

Dimension

```
<dimen name="…">dimension</dimen>
```

dimension ist eine Kommazahl gefolgt von der Maßeinheit (dp, sp, pt, px, mm, in).

```
<dimen name="schriftgroesse">16sp</dimen>
int schriftGroesse = getResources().getDimension(R.id.schriftgroesse);
```

INFO

Der Wert wird abhängig von der Maßeinheit und der Bildschirmmetrik ermittelt. Wenn wir also geräteunabhängige Maßeinheiten verwenden, erhalten wir auf jedem Gerät den entsprechenden Wert für die Schriftgröße, damit die Schrift überall gleich aussieht.

Das ist eine gute Sache, um sich selbst die Umrechnung zu sparen.

ID

```
<item type="id" name="id_name"/>
<item type="id" name="about_dialog"/>
int id = R.id.about_dialog;
```

Integer

```
<integer name="…">ganzzahliger Wert</integer>
<integer name="anzahl_spieler">5<integer>
int wert = getResources().getInteger(R.id.anzahl_spieler);
```

Integer-Array

```
<integer-array name="…">
    <item>ganzzahliger Wert 1</item>
    <item>ganzzahliger Wert 2</item>
    <item>ganzzahliger Wert n</item>
</integer-array>

<integer-array name="prioritaeten">
    <item>1</item>
    <item>2</item>
    <item>3</item>
</integer-array>
int prios[]= getResources().getIntArray(R.id.prioritaeten);
```

String

```
<string name="…">Zeichenkette</string>
<string name="begruessung">Hallo!</string>
<string name="persoenliche_begruessung">Hallo %s!</string>
```

```
<string name="ausfuehrliche_begruessung">Hallo %1$s! Guten %2$s, wie geht es
%3$s</string>
String begruessung = getResources().getString(R.id.begruessung);
String pbegruessung = getResources().getString(R.id.persoenliche_
begruessung,name);
String ausfuehrlicheBegruessung = getResources().getString(R.
id.ausfuehrliche_begruessung,name,tagsezeit,duodersie);
```

String-Array

```
<string-array name="…">
    <item>Zeichenkette 1</item>
    <item>Zeichenkette 2</item>
    <item>Zeichenkette n</item>
</string-array>

<string-array name="kategorien">
    <item>Einkaufsliste</item>
    <item>Bücherwunsch</item>
    <item>Tolle Weine</item>
</string-array>
String[] kategorien = getResources().getStringArray(R.id.kategorien);
```

Plurals

```
<plurals name="…">
<item quantity=["zero" | "one" | "two" | "few" | "many" |
"other"]>Zeichenkette</item>
</plurals>
```

Beispiel für die Anzahl gefundener Lieder. Standard-XML-Datei in `res/values/strings.xml` für die Standardsprache:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <plurals name="numberOfSongsAvailable">
        <item quantity="zero">Keine Lieder gefunden.</item>
        <item quantity="one">Ein Lied gefunden.</item>
        <item quantity="other">%d Lieder gefunden.</item>
    </plurals>
</resources>
```

Die polnische Übersetzung in `res/values-pl/strings.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <plurals name="numberOfSongsAvailable">
        <item quantity="one">Znaleziono jedną piosenką.</item>
        <item quantity="few">Znaleziono %d piosenki.</item>
        <item quantity="other">Znaleziono %d piosenek.</item>
    </plurals>
</resources>
```

Die beiden Beispiele sind der Dokumentation auf http://developers.android.com entliehen. Die Umsetzung ins Deutsche habe ich noch um den Aspekt »Keine Lieder gefunden« erweitert, das konnte ich aber mangels Sprachkenntnisse in der polnischen Variante nicht machen. Ich hoffe auch, dass die polnische Übersetzung überhaupt stimmt.

```
int songs = songsProvider.getCount();
String songsDisplay = getResouce().getQuantityString(R.id.
numberOfSongsAvailable,songs);
```

Und hier noch ein Beispiel für Formatangaben in Plurals:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <plurals name="numberOfSongsAvailable">
        <item quantity="zero">Keine Lieder in Kategorie %s gefunden.</item>
        <item quantity="one">Ein Lied in Kategorie %1 gefunden.</item>
        <item quantity="other">%d Lieder in Kategorie %s gefunden.</item>
    </plurals>
</resources>
String kategorie = getKategorie();
int songs = songsProvider.getCount(kategorie);
String songsDisplay = getResouce().getQuantityString(R.id. numberOfSongsAvai
lable,songs,kategorie);
```

Typed Array

```
<array name="…">
    <item>Referenz auf Resource oder einfacher Datentyp 1</item>
    <item>Referenz auf Resource oder einfacher Datentyp 2</item>
    <item>Referenz auf Resource oder einfacher Datentyp n</item>
</array>
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <array name="icons">
        <item>@drawable/home</item>
        <item>@drawable/settings</item>
        <item>@drawable/logout</item>
    </array>
    <array name="colors">
        <item>#FFFF0000</item>
        <item>#FF00FF00</item>
        <item>#FF0000FF</item>
    </array>
</resources>
Resources res = getResources();
TypedArray icons = res.obtainTypedArray(R.array.icons);
Drawable drawable = icons.getDrawable(0);

TypedArray colors = res.obtainTypedArray(R.array.icons);
int color = colors.getColor(0,0);
```

3.10.6 Komplexe Ressourcen

Neben den einfachen Ressourcen stehen komplexere Ressourcen für die Deklaration von Layouts, der Verwaltung von Drawables, der Deklaration von Menüs, Animationen und Farblisten zur Verfügung. Hier wollen wir uns nur kurz die Grundstruktur anschauen, wie die Ressourcen verwendet werden sehen wir dann später bei der Behandlung der jeweiligen Themen.

Animation Resources

Mit *Animation Resources* kann man Elemente der Benutzeroberfläche animieren, z.B. rotieren, verschieben, skalieren oder die Transparenz verändern. Diese Art der Animation nennt sich *Tween Animation,* da sich bestimmte Objekteigenschaften, z.B. der Position, von einem Ursprungswert zu einem Zielwert über einen Pfad verändern. Der Pfad wird hier durch Interpolatoren bestimmt, die die Veränderung der Objekteigenschaft über die Zeit berechnet. Es gibt verschiedene Interpolatoren, die z.B. eine langsame Beschleunigung oder ein Abbremsen oder die ein Overshoot, also das Überschreiten eines Maximalwerts und dann ein Zurückschnellen zum Maximalwert, simulieren.

Durch die Kombination der unterschiedlichen Transformationen mit unterschiedlichen Interpolatoren lassen sich interessante Effekte auf den Views erreichen.

Eine weitere Möglichkeit ist, ein Drawable als *Frame Animation* zu deklarieren. Eine Frame Animation ist wie ein Daumenkino, bei der Einzelbilder nacheinander in einer gewissen Geschwindigkeit abgespielt werden.

Diese `AnimationDrawable`-Objekte können, da sie ganz »normale« Drawables darstellen, überall dort benutzt werden, wo Drawables zum Einsatz kommen, z.B. in einer ImageView oder auch als Hintergrund-Drawable für andere Views.

Ab Android 3.0 ist das View-Animation-System schon dabei zu »veralten«. Mit Android 3.0 wurde ein Property-Animation-Framework eingeführt, das die Animation beliebiger Objekte und deren Eigenschaften erlaubt.

Da es aber durchaus noch eine Weile dauern kann, bis die meisten Geräte mit Android 3 und höher ausgeliefert worden sind, sollte man diese Form der Animation für Views auf alle Fälle noch berücksichtigen. Wenn wir allerdings dezidiert für Android 3-Geräte entwickeln, ist das Property-Animation-Framework der spezialisierten View-Animation vorzuziehen.

Jede Animation-Ressource wird in einem eigenen XML-File abgelegt.

Die Ressourcen-ID einer Animation wird aus dem Dateinamen der XML-Datei gebildet.

Listing 3.9: **Animation Resources – Grundstruktur Tween Animation**

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@[package:]anim/interpolator_resource"
android:shareInterpolator=["true" | "false"] >
    <alpha
```

```
    android:fromAlpha="float"
    android:toAlpha="float" />
    <scale
    android:fromXScale="float"
    android:toXScale="float"
    android:fromYScale="float"
    android:toYScale="float"
    android:pivotX="float"
    android:pivotY="float" />
    <translate
    android:fromXDelta="float"
    android:toXDelta="float"
    android:fromYDelta="float"
    android:toYDelta="float" />
    <rotate
    android:fromDegrees="float"
    android:toDegrees="float"
    android:pivotX="float"
    android:pivotY="float" />
    <set>
    ...
    </set>
</set>
```

Listing 3.10: **Grundstruktur Frame-Animation**

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
android:oneshot=["true" | "false"] >
    <item
    android:drawable="@[package:]drawable/drawable_resource_name"
    android:duration="integer" />
</animation-list>
```

Die View-Animationen werden meist über `View.setAnimation(<Resource-ID>)` benutzt oder mittels `AnimatorInflater.loadAnimator(context, <Resource-ID>)` geladen.

Die Property-Animationen können ebenfalls in XML-Dateien abgelegt und daraus geladen werden.

Listing 3.11: **Property-Animation in XML**

```
<set android:ordering="sequentially">
    <set>
        <objectAnimator
        android:propertyName="x"
        android:duration="500"
        android:valueTo="400"
        android:valueType="int"/>
        <objectAnimator
        android:propertyName="y"
        android:duration="500"
        android:valueTo="300"
        android:valueType="int"/>
    </set>
```

```
    <objectAnimator
    android:propertyName="alpha"
    android:duration="500"
    android:valueTo="0f"/>
</set>
```

Mittels `AnimatorInflater.loadAnimator(context, <Resource-ID>)` kann eine solche XML-Ressource geladen werden.

INFO

Die Ressourcen der Property-Animation werden per Definition im Verzeichnis `res\animator` abgelegt, die Ressourcen der View Animation in `res\anim`. Es ist aber kein Fehler, wenn man Property Animatoren ebenfalls in `res\anim` ablegt. Benutzt man das Eclipse-Plug-in, legt das Plug-in die Dateien von sich aus an die richtige Stelle.

Color State Lists

Color State Lists liefern die Farbe für die unterschiedlichen Status einer View. Folgende Status kann eine View annehmen:

android:state_pressed	Die View wurde gedrückt.
android:state_focsed	Die View besitzt den Eingabefokus.
android:state_selected	Die View ist selektiert (z.B. ein ListItem).
android:state_checkable	Die View kann angehakt werden (z.B. eine CheckBox oder ein RadioButton).
android:state_checked	Die View ist angehakt.
android:state_enabled	Die View ist aktiv, d.h. kann angeklickt, berührt, gedrückt, was auch immer werden.
android:state_window_focused	Das Fenster zu dem die View gehört hat den Fokus.

Tabelle 3.11: **Die möglichen Status einer View**

Die unterschiedlichen Status müssen dem Benutzer angezeigt werden, das geschieht z.B. über verschiedene Farben, und verschiedene Hintergründe.

ACHTUNG

Die Color State List ist kein `Drawable`! Mit den *Color State Lists* kann man nur Farbwerte setzen. Der Hintergrund von Views muss ein `Drawable` sein. Es gibt aber mit dem `StateListDrawable` einen äquivalenten Typ für Drawables, und auch eine entsprechende Ressource kann angelegt werden.

Die Color State List wird in einem eigenen XML-File abgelegt, der Filename wird zur Ressourcen-ID.

Listing 3.12: **Grundstruktur Color State List**

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android" >
    <item
    android:color="hex_color"
    android:state_pressed=["true" | "false"]
    android:state_focused=["true" | "false"]
    android:state_selected=["true" | "false"]
    android:state_checkable=["true" | "false"]
    android:state_checked=["true" | "false"]
    android:state_enabled=["true" | "false"]
    android:state_window_focused=["true" | "false"] />
</selector>
```

Drawables

Es gibt einen ganzen Strauß von Drawables. Die einfachste Form sind Bitmaps, über 9-Patch-Dateien bis hin zu `StateListDrawables` oder `TransitionDrawables`.

Den Drawables werden wir einen eigenen Abschnitt widmen.

Layouts

Mit den Layouts kommen wir zwangsweise als Erstes in Berührung wenn wir eine Activity anlegen, denn der Projekterstellungsassistent legt für diese Activity ein Layout an.

Den Layouts widmen wir ebenfalls einen eigenen Abschnitt.

Menus

In den Menu-Ressourcen werden, nun ja, wie der Name schon sagt, Auswahlmenüs abgelegt.

Es gibt drei Menü-Arten:

1. Option Menu
2. Context Menu
3. Submenu

und unter Android 3 werden noch die *Action Items* eingeführt, die in der *Action Bar* erscheinen und so einen schnellen Zugriff auf die wichtigsten Menüoptionen bieten.

Listing 3.13: **Grundstruktur Menu-Ressource**

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<item android:id="@[+][package:]id/resource_name"
  android:title="string"
  android:titleCondensed="string"
  android:icon="@[package:]drawable/drawable_resource_name"
```

```
  android:onClick="method name"
  android:showAsAction=["ifRoom" | "never" | "withText" | "always"]
  android:actionLayout="@[package:]layout/layout_resource_name"
  android:actionViewClass="class name"
  android:alphabeticShortcut="string"
  android:numericShortcut="string"
  android:checkable=["true" | "false"]
  android:visible=["true" | "false"]
  android:enabled=["true" | "false"]
  android:menuCategory=["container" | "system" | "secondary" | "alternati➥
    ve"]
  android:orderInCategory="integer" />
<group android:id="@[+][package:]id/resource name"
   android:checkableBehavior=["none" | "all" | "single"]
   android:visible=["true" | "false"]
   android:enabled=["true" | "false"]
   android:menuCategory=["container" | "system" | "secondary" | "alternati➥
    ve"]
   android:orderInCategory="integer" >
<item />
</group>
<item >
<menu>
  <item />
</menu>
</item>
</menu>
```

Im Zusammenhang mit den Activities und dem Userinterface werden wir uns mit den Menüs noch eingehender beschäftigen.

Styles

Styles bieten die Möglichkeit, das Layout unserer Benutzeroberfläche strikt vom Aussehen der einzelnen Elemente (Farbe, Hintergrund, Schriftgröße) zu trennen. Sie Styles sind mit Cascading Style Sheets aus dem Web-Design vergleichbar, sie bieten hier ebenfalls die Möglichkeit, Stile zu vererben. Die Styles erhalten einen Namen und können mit anderen Stilen innerhalb einer XML-Ressourcendatei zusammengefasst werden.

Listing 3.14: **Grundstruktur einer Style-Ressource**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<style
    name="style_name"
    parent="@[package:]style/style_to_inherit">
    <item
    name="[package:]style_property_name"
    >style_value</item>
</style>
<style
    name="style_name_2"
    parent="@[package:]style/style_to_inherit">
    <item
```

```
    name="[package:]style_property_name"
    >style_value</item>
</style>
[…]
</resources>
```

Ein Stil kann dann von anderen Ressourcen über den Namen referenziert werden:

```
<TextView style="@style/<style_name>" …/>
```

Styles werden auch dazu verwendet, Themes zu generieren. In den Themes sind Items mit Namen zusammengefasst, die wiederum in Styles oder den entsprechenden Attributen benutzt werden können.

Wendet obiges Konstrukt den Stil `<style_name>` aus unseren Stilressourcen an, so könnten wir auch mittels:

```
<TextView textColor="?<style_property_name>" …/>
```

auf die entsprechende Stilangabe in `<item name="<style_property_name>">…<item>` zugreifen.

Themes können einer ganzen Anwendung oder einer Activity mittels `android:theme="@style/<style_name>"` zugewisen werden.

Die Deklaration von

```
<activity android:theme="@android:style/Theme.Translucent">
```

für unsere Activity wählt ein Theme aus, bei dem der Fensterhintergrund transparent ist.

TIPP

Die Verwendung von Styles und Themes ist ratsam, da man dadurch ein konsistentes Look and Feel der Applikationen erreicht. Statt die Textgrößen fest einzustellen, sollte man auf die entsprechenden Attribute des Themes zugreifen, z.B. `<TextView android:textSize="?android:textAppearanceLarge" …/>`**.**

Kleine Zwischenbilanz

Wir haben eine Menge über das Ressourcensystem gelernt. Wir sollten möglichst viel von dem Ressourcensystem nutzen und möglichst wenig dynamisch in der Anwendung generieren. Das ist vor allem bei den Layouts, aber auch bei der Verwendung von Zeichenketten und Dimensionsangaben wichtig, um eine hohe Konfigurationsunabhängigkeit zu erreichen und die Internationalisierung unserer Anwendung zu erleichtern.

Es ist etwas aufwendiger, jeden String zuerst in den Ressourcen anzulegen, aber die Wiederverwendbarkeit und die leichtere Umsetzbarkeit auf andere Sprachen entschädigen für den Mehraufwand. Nicht zuletzt das dröge Ändern von Schreibfehlern oder die Anpassung von Farben, weil uns doch was nicht grün genug ist, werden durch die konsequente Ressourcennutzung extrem vereinfacht.

Wenden wir uns nun dem Userinterface zu.

3.11 Das Userinterface

Natürlich kommt es auf die inneren Werte an, aber eine schöne und funktionale Oberfläche ist doch auch sehr wichtig. Bevor wir mit Activities starten können müssen wir uns ein wenig anschauen, wie die Oberfläche mit Android erstellt wird, welche Elemente wir zur Verfügung haben und was uns das System sonst noch alles bietet.

Wie wir im vorherigen Abschnitt gelernt haben, wird auch die Benutzeroberfläche am besten per Ressourcen erstellt und nicht dynamisch in Java programmiert.

Natürlich kann auch das für eigene Komponenten oder bei speziellen dynamischen Oberflächen sinnvoll sein, aber soweit es geht, ist die Deklaration der Oberfläche in den Ressourcen vorzuziehen.

3.11.1 Wichtige UI-Elemente

Die Aufteilung der UI-Elemente erfolgt in:

1. Views (android.view.View)
2. View Groups (android.view.ViewGroup)

Eine ViewGroup ist selbst wieder eine View, kann aber weitere Views beinhalten. Das ergibt dann eine View-Hierarchie. Die äußeren Elemente können wir auch als Container bezeichnen, die wiederum Elemente, seine Kindelemente, beinhalten.

Eine Ausprägung einer ViewGroup sind Layout-Elemente. Das Layout-Element stellt selbst nichts auf dem Bildschirm dar, es beinhaltet aber View-Elemente, die es in bestimmter Art und Weise anordnet, also »layoutet«.

Eine andere Ausprägung der ViewGroups ist z.B. die Gallery.

View-Elemente wie Buttons, Textfelder, Eingabefelder, Listen werden Widgets genannt. Widgets sind die Elemente, mit denen der Benutzer etwas machen kann oder die dem Benutzer etwas zeigen.

Man kann Benutzeroberflächen auf zwei Arten erstellen:

1. Per XML-Deklaration als Ressourcendatei in `res/layout`
2. Zur Laufzeit innerhalb unseres Java-Programms mittels direkter Nutzung der Klassen

Es ist, wie im Abschnitt über die Ressourcen beschrieben, empfehlenswert, so viel wie möglich über die XML-Deklaration abzubilden und die Erstellung der Oberfläche zur Laufzeit nur dann zu nutzen, wenn es unbedingt nötig ist.

Es ist möglich, eigene Widgets oder View-Elemente zu erstellen, die auch in den XML-Deklarationen verwendet werden können.

3.11.2 Layouts definieren

Die XML-Deklaration von Layouts erfolgt in XML-Dateien, die im Verzeichnis `res/layout` bzw. in konfigurationsspezifischen Verzeichnissen angelegt werden.

Die Layout-Definition hat folgende allgemeine Syntax:

Listing 3.15: **Allgemeiner Aufbau einer Layout-Resource**

```
<?xml version="1.0" encoding="utf-8"?>
<ViewGroup xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@[+][package:]id/resource_name"
    android:layout_height=["dimension" | "match_parent" | "wrap_content"]
    android:layout_width=["dimension" | "match_parent" | "wrap_content"]
    [ViewGroup-specific attributes] >
    <View
        android:id="@[+][package:]id/resource_name"
        android:layout_height=["dimension" | "match_parent" | "wrap_con➥
          tent"]
        android:layout_width=["dimension" | "match_parent" | "wrap_content"]
        [View-specific attributes] >
        <requestfocus/>
    </View>
    <ViewGroup >
        <View />
    </ViewGroup>
    <include layout="@layout/layout_resource"/>
</ViewGroup>
```

Das oberste Element einer Layout-Ressource ist immer eine View-Group, in der wiederum verschiedene Views und weitere View-Groups deklariert werden.

Statt der hier angegebenen Basisklassen `ViewGroup` und `View` setzen wir die Klassennamen der abgeleiteten Views ein wie z.B. `LinearLayout` und `TextView`.

Listing 3.16: **Beispiel einer Layout-Ressource**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <EditText
    android:id="@+id/nfcforegrounddipsatch_edittext"
    android:hint="@string/nfcforegrounddipsatch_edittext_hint"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    </EditText>
    <Button
    android:text="@string/nfcwritetotag"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/cmdwritetotag">
    </Button>
```

```
    <TextView
    android:id="@+id/nfcforegrounddipsatch_text"
    android:textSize="16dp"
    android:textStyle="bold"
    android:text=""
    android:gravity="center"
    android:layout_height="match_parent"
    android:layout_width="match_parent">
    </TextView>
</LinearLayout>
```

Die folgenden Attribute sind für alle View-Groups und Views gültig:

android:id="@[+][package:]id/resource_name"	Eindeutige ID des View-/View-Group-Elements
android:layout_height=["dimension" | "match_parent" | "wrap_content"]	Höhe des Elements
android:layout_width=["dimension" | "match_parent" | "wrap_content"]	Breite des Elements

Tabelle 3.12: **Allgemeine Attribute für View- und ViewGroup-Klassen**

Über die ID eines Elements kann man später im Programm auf das Element in der View-Hierarchie zugreifen:

```
TextEdit textEdit = (TextEdit)findViewById(R.id.nfcforegrounddipsatch_edit-
text);
```

Oder z.B. ermitteln, dass der Button mit der ID `R.id.cmdwritetotag` gedrückt wurde.

Die ID ist auch bei Verwendung des Layout-Typs RelativeLayout wichtig. In einem `RelativeLayout` kann man Views relativ zueinander ausrichten. Dabei beziehen sich die einzelnen Elemente durch ihre ID aufeinander.

Mit `android:layout_width` und `android:layout_height` bestimmt man die Größe des Elements innerhalb des Containers. Neben einer expliziten Maßangabe, die die Größe festlegt, spielen die Werte `MATCH_PARENT` und `WRAP_CONTENT` eine wichtige Rolle, um flexible Layouts zu bauen.

`MATCH_PARENT` bestimmt, dass die Höhe oder Breite die maximale Ausdehnung in der jeweiligen Richtung im Container einnimmt. Im obigen Beispiel wird der lineare Layout-Container auf der obersten Ebene mit

```
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
```

definiert. Dadurch nimmt das Layout den verfügbaren Platz auf dem Bildschirm ein und organisiert die Kindelemente von oben nach unten, egal in welcher »Lage« sich der Bildschirm befindet. Wird der Bildschirm gedreht, so passt sich die View-Hierarchie automatisch an die neuen Abmessungen an.

Das Texteingabefeld wiederum wird mit

```
android:layout_width="match_parent"
android:layout_height="wrap_content"
```

definiert. Dadurch wird das Eingabefeld so breit wie der verfügbare Platz im Layout-Container. Da dieser selbst so groß wird wie der Bildschirm breit ist, nimmt das Eingabefeld also die Breite des Bildschirms in Anspruch. In die Berechnung fließen allerdings noch ggf. ein Padding (Auspolsterung) und Margins (Ränder) mit ein.

Durch die Angabe von `WRAP_CONTENT` wird das Element so groß, dass es seine Unterelemente alle darstellen kann, maximal aber so groß wie das äußere Element wiederum es zulässt, was je nach Layout/View dazu führen kann, dass die Inhalte gescrollt werden müssen/können, wenn sie mehr Platz benötigen würden.

In unserem Beispiel ist das Texteingabefeld mit der Höhe `WRAP_CONTENT` definiert. Dadurch wird es so hoch, dass es seinen Inhalt darstellen kann. Bei einem Texteingabefeld ist das mindestens eine Zeile Text unter der Berücksichtigung der gewählten Schriftgröße und einem vorgegebenen Padding.

Neben den allgemeinen Attributen führen die unterschiedlichen Views weitere spezielle Attribute ein, die das Erscheinungsbild und das Verhalten der View bestimmen. Das `LinearLayout` besitzt z.B. ein Attribut `android:orientation` mit dem die Richtung des Layouts, horizontal oder vertikal, bestimmt wird.

Alle für das »Layouten« eines Elements relevanten Attribute beginnen mit `android:layout_`*`irgendwas`*, sofern es sich um durch das Android-SDK definierte Klassen handelt. Diese Attribute bestimmen immer, wie sich das Element in seinem äußeren Container verhält. Der äußere Container berücksichtigt also die Attribute seiner innen liegenden Elemente. Das `LinearLayout` bezieht die Ausrichtung der Kindelemente aus seinem Attribut `android:orientation`. Welchen Platz die Kindelemente für sich beanspruchen wollen, ermittelt das `LinearLayout` aus den Attributen `android:layout_width` und `android:layout_height`.

Die einer View zugrunde liegende Geometrie ist ein Rechteck. Die View hat eine Position innerhalb ihres umschließenden Rechtecks sowie eine Breite und eine Höhe. Die Angaben der Position beziehen sich immer relativ zum umschließenden Rechteck, also dem Platz, den der Container für sich beansprucht.

Die Position einer View bestimmen wir nie fest, die Position orientiert sich an den Layoutvorgaben des Elements innerhalb des umgebenden Layouts. Wir können per `android:layout_width` und `android:layout_height` festlegen, wie viel Platz das Element beanspru-

chen soll. Weiterhin können wir mittels `android:paddingLeft`, `android:paddingTop`, `android:paddingRight` und `android:paddingBottom` definieren, wie viel Auspolsterung um den Inhalt des Elements gelegt wird. Mittels der Layout-Attribute `android:layout_marginLeft`, `android:layout_marginTop`, `android:layout_marginRight` und `android:layout_marginBottom` kann ein Element innerhalb eines Layouts einen Rand um das Element definieren.

INFO

Die Auspolsterung (Padding) ist immer Bestandteil des Elements. Durch die Auspolsterung wird nur der Bereich, in dem das Element seinen Inhalt darstellt, entsprechend verkleinert. Der Rand ist nicht Bestandteil des Elements, sondern definiert die Abstände der Elemente in einem Layout-Container von den Rändern des Containers respektive zu umgebenden Elementen.

Allen Views gemeinsam sind noch einige Attribute, die das Aussehen und zum Teil auch das Verhalten der View bestimmen. Diese Attribute können für alle abgeleiteten View-Klassen benutzt werden.

android:alpha
setAlpha(float)
	Definiert die Transparenz (Alpha-Kanal) der View. 0 ist komplett durchsichtig, 1 ist vollkommen undurchsichtig. Achtung: Das wirkt sich auch auf den Inhalt der View aus, nicht nur auf den Hintergrund der View. Kann z.B. mit dem Property-Animation-Framework benutzt werden, um eine View auszublenden oder einzublenden.
	`android:alpha="0.5"`
	`view.setAlpha(0.5);`

android:background
setBackgroundResource(int)
	Drawable für den Hintergrund. **Wichtig**: Das kann auch eine Referenz auf eine Farbe sein, und für Hintergründe, die sich dynamisch an die Größe anpassen sollen, verwendet man ein 9-Patch-Bild.
	`android:background="@drawable/playground_background.png"`
	`view.setBackgroundResource(R.id.playground_background);`
	`android:background="@color/blauer_hintergrund"`
	`view.setBackgroundResource(R.id.blauer_hintergrund);`
	Wenn wir einen transparenten Hintergrund benötigen, z.B. wenn wir die View über eine andere legen wollen, dann benutzen wir android:background="@android:color/transparent"

Tabelle 3.13: **Übersicht über die allgemeinen Attribute von Views**

android:clickable
setClickable(boolean)
	Bestimmt ob die View auf Klicks reagiert. Wenn diese Eigenschaft auf true gesetzt wird dann ändert sich der Status der View auf »Pressed« bei jedem Klick des Benutzers in die View.
	`android:clickable="true"`
	`view.setClickable(true);`

android:contentDescription
setContentDescription (CharSequence)
	Beschreibung der View. Sollte benutzt werden um die View zu beschreiben, und wird durch die Zugangsunterstützung (Accessibility-Support) ausgewertet. Die Text-to-Speech Unterstützung kann dann z.B. vorlesen, was ein reiner ImageButton bedeutet, der selbst keinen Text bereitstellt.
	`android:contentDescription="@string/neuer_eintrag_beschreibung"`
	`view.setContentDescription(getResources().getString(R.id. neuer_eintrag_beschreibung);`

android:duplicateParentState	Wenn diese Eigenschaft auf »true« gesetzt ist, dann wird der Status (Focused, Enabled, Selected, Pressed, Window Focused) des übergeordneten Elements auf dieses Element übernommen.
android:fadingEdge = "none | horizontal | vertical" setVerticalFadingEdgeEnabled (boolean) setHorizontalFadingEdgeEnabled (boolean)	Legt fest, welche Ecke der View beim Scrollen verblassen soll. Damit erreicht man den Effekt, dass die beiden Enden des Scrollbereiches während des Scrollens verschwimmen.
android:fadingEdgeLength getVerticalFadingEdgeLength() getHorizontalFadingEdgeLength()	Legt die Größe des Bereichs fest der beim Scrollen verblassen soll.
android:filterTouchesWhenObscured setFilterTouchesWhenObscured(boolean)	Legt fest, ob Touch-Events verarbeitet werden sollen, wenn die View durch etwas anderes verdeckt wird. Wird zur Erhöhung der Sicherheit benutzt, damit der Anwender nicht versehentlich eine Funktion auslösen kann, während etwas anderes im Vordergrund zu sehen ist.
android:focusable setFocusable(boolean)	Legt fest, ob die View den Fokus erhalten kann. Diese Eigenschaft ist wichtig, denn wenn eine View den Fokus nicht erhalten kann, werden auch keine Eingabeereignisse an die View geschickt.

Tabelle 3.13: **Übersicht über die allgemeinen Attribute von Views (Forts.)**

android:focusableInTouchMode setFocusableInTouchMode (boolean)	Legt fest, ob die View den Fokus erhält, wenn der Anwender per Touchscreen mit der View interagiert. Texteingabefelder z.B. erhalten den Fokus, wenn man hineintippt (android:focusableInTouchMode="true"), Buttons hingegen erhalten im Touchmode den Fokus nicht (android:focusableInTouchMode="false"), sondern feuern nur das entsprechende Klick-Ereignis ohne den Fokus zum Button zu bewegen. Der Fokus verbleibt stattdessen beim aktuell fokussierten Element.
android:hapticFeedbackEnabled setHapticFeedbackEnabled (boolean)	Legt fest, ob die View eine haptische Rückmeldung auslöst, wenn sie angetippt wird. Dazu wird in der Regel der Vibrator mit unterschiedlichen Vibrationsmustern benutzt, um das Ereignis zu quittieren.
android:id setId(int)	ID für die View, um sie später mit View.findViewById() oder Activity.findViewById() aus der View-Hierarchie zu ermitteln und im Programm auf die View zuzugreifen. Ebenfalls wichtig, wenn man Views im RelativeLayout anordnen will.
android:isScrollContainer	Legt fest, ob die View ihren Inhalt scrollen kann, und damit verkleinert werden kann wenn die virtuelle Tastatur aktiviert wird.
android:keepScreenOn setKeepScreenOn(boolean)	Legt fest, ob der Bildschirm aktiv bleiben soll, solange die View aktiv ist. Das kann z.B. bei Views sinnvoll sein, die über einen längeren Zeitraum etwas anzeigen und bei denen wir verhindern wollen, dass man immer mal wieder hineintippen muss, um die Anwendung aufzuwecken.
android:longClickable setLongClickable(boolean)	Legt fest, ob die View zwischen kurzen und langen Toch-Events unterscheiden soll. Lange Klicks werden z.B. verwendet, um ein Kontextmenü aufzurufen.
android:minHeight	Legt die gewünschte minimale Höhe fest. Ob die View diese Breite mindestens erhält, wird allerdings nicht garantiert und hängt vom umgebenden Container ab.
android:minWidth	Legt die gewünschte minimale Breite fest. Ob die View diese Breite mindestens erhält, wird allerdings nicht garantiert und hängt vom umgebenden Container ab.

Tabelle 3.13: **Übersicht über die allgemeinen Attribute von Views (Forts.)**

android:nextFocusDown setNextFocusDownId(int)	Hiermit wird die Reihenfolge der Sprünge durch die View-Hierarchie bestimmt, indem wir jeweils die ID des folgenden, vorangehenden, rechtsseitig oder linksseitig befindlichen View-Elements angeben. Das kann bei komplexen Eingabeformularen wichtig sein, damit die Bedienung über D-Pads, Trackballs oder auch Cursor-Tasten (falls die vorhanden sind) in logischer Reihenfolge funktioniert.
android:nextFocusForward setNextFocusForwardId(int)	
android:nextFocusLeft setNextFocusLeftId(int)	
android:nextFocusRight setNextFocusRightId(int)	
android:nextFocusUp setNextFocusUpId(int)	

android:onClick	Methode im Kontext der View (innerhalb der umgebenden Activity), die beim Klick aufgerufen werden soll. Beispiel:
	`android:onclick="machwas"`
	`[…]`
	`class MeineActivity : public Activity`
	`{`
	`[…]`
	`    public void machwas(View view)`
	`    {`
	`    }`
	`[…]`
	`}`
	Ich empfehle allerdings, darauf zu verzichten, und stattdessen einen Listener zu benutzen.

android:padding setPadding(int,int,int,int)	Legt die Auspolsterung (Abstand des Inhalts zum Rand der View) fest.
android:paddingBottom setPadding(int,int,int,int)	
android:paddingLeft setPadding(int,int,int,int)	
android:paddingRight setPadding(int,int,int,int)	
android:paddingTop setPadding(int,int,int,int)	
android:rotation setRotation(float)	Drehung der View in Grad um den Pivot-Punkt. Kann z.B. in Verbindung mit dem Property-Animation-Framework verwendet werden.

Tabelle 3.13: **Übersicht über die allgemeinen Attribute von Views (Forts.)**

android:rotationX setRotationX(float)	Drehung der View um die jeweilige Achse, wird für 3D-Effekte benutzt. Kann z.B. in Verbindung mit dem Property-Animation-Framework verwendet werden.
android:rotationY setRotationY(float)	
android:saveEnabled setSaveEnabled(boolean)	Wenn hier false angegeben wird, dann wird der Zustand der View nicht gespeichert, falls die Anwendung schlafen gelegt wird.
android:scaleX setScaleX(float)	Skalierungsfaktor in X- bzw. Y-Richtung Kann z.B. in Verbindung mit dem Property-Animation-Framework verwendet werden.
android:scaleY setScaleY(float)	
android:scrollX	Scroll-Offset in X- und Y-Richtung.
android:scrollY	
android:scrollbarAlwaysDraw-HorizontalTrack	Legt fest, ob die horizontale Scrollbar immer sichtbar ist
android:scrollbarAlwaysDraw-VerticalTrack	Legt fest, ob die vertikale Scrollbar immer sichtbar ist
android:scrollbarDefaultDelay-BeforeFade	Legt die Millisekunden fest, bis die Scrollbars ausgeblendet werden.
android:scrollbarFadeDuration	Legt die Dauer des Ausblendevorgangs in Millisekunden fest.
android:scrollbarSize	Höhe der horizontalen bzw. Breite der vertikalen Scrollbar
android:scrollbarStyle = " insideOverlay | insideInset | outsideOverlay | outsideInset"	Legt fest, wo die Scrollbars angeordnet werden: inside: Innerhalb der View, im Padding-Bereich outside: An der Kante der View Overlay: Überlagert den Inhalt/die Kante Inset: Ist im Inhalt/an der Kante eingefügt
android:scrollbarThumbHorizontal	Hier kann man jeweils für den Thumb (der Anfasser) bzw. den Balken der Scrollbar ein Drawable angeben.
android:scrollbarThumbVertical	
android:scrollbarTrackHorizontal	
android:scrollbarTrackVertical	
android:scrollbars="none | horizontal | vertical"	Welche Scrollbars sollen angezeigt werden?

Tabelle 3.13: **Übersicht über die allgemeinen Attribute von Views (Forts.)**

android:soundEffectsEnabled setSoundEffectsEnabled (boolean)	Ähnlich wie haptisches Feedback, aber als Soundeffekte.
android:tag	Damit kann man der View einen Namen verpassen und dann mittels View.getTag() den Namen auslesen bzw. mit View.findViewWith-Tag() eine View anhand des Namens heraus-suchen.
android:transformPivotX setPivotX(float)	Pivot-Koordinaten auf die sich die Transfor-mationen wie Rotation, Translation und Skalierung beziehen. Wichtig bei View-Animationen.
android:transformPivotY setPivotY(float)	
android:translationX setTranslationX(float)	Horizontale bzw. vertikale Verschiebung der View. Kann z.B. in Verbindung mit dem Property-Animation-Framework verwendet werden.
android:translationY setTranslationY(float)	
android:visibility="visible | invisible | gone" setVisibility(int)	Legt die Sichtbarkeit der View fest: visible/View.VISIBLE: Sichtbar. invisible/View.INVISIBLE: Unsichtbar, aber der Platz bleibt reserviert. gone/View.GONE: Unsichtbar, und der einge-nommene Platz verschwindet ebenfalls.

Tabelle 3.13: **Übersicht über die allgemeinen Attribute von Views (Forts.)**

Mit diesen allgemeinen Attributen können alle Views gestaltet werden, die spezialisierten Views bringen dann weitere Attribute mit, z.B. für Schriftart und Größe und vieles anderes mehr.

Wenn wir für eine Activity ein Benutzerinterface erstellen, legen wir in der Regel immer erst das Layout fest und füllen dieses Layout mit unseren Widgets. Glücklicherweise bietet die Integration in die Eclipse bereits einige Hilfsmittel, um schnell solche Layouts zu erstellen.

3.11.3 Anlegen von Layouts in Eclipse

Wir wollen dann direkt mal in die Vollen gehen und uns Beispiel-Layouts für unsere Spiel-wiese anlegen, die wir dann später verwenden können.

Zum Anlegen von Layout, aber auch aller anderen Ressourcen, benutzen wir den Android XML-File-Wizard.



Abbildung 3.11: **Anlegen von Layouts und anderer Ressourcen**

Der folgende Dialog bietet uns alle Möglichkeiten, die wir zur Verwaltung der Ressourcen in unserem Android-Projekt benötigen.

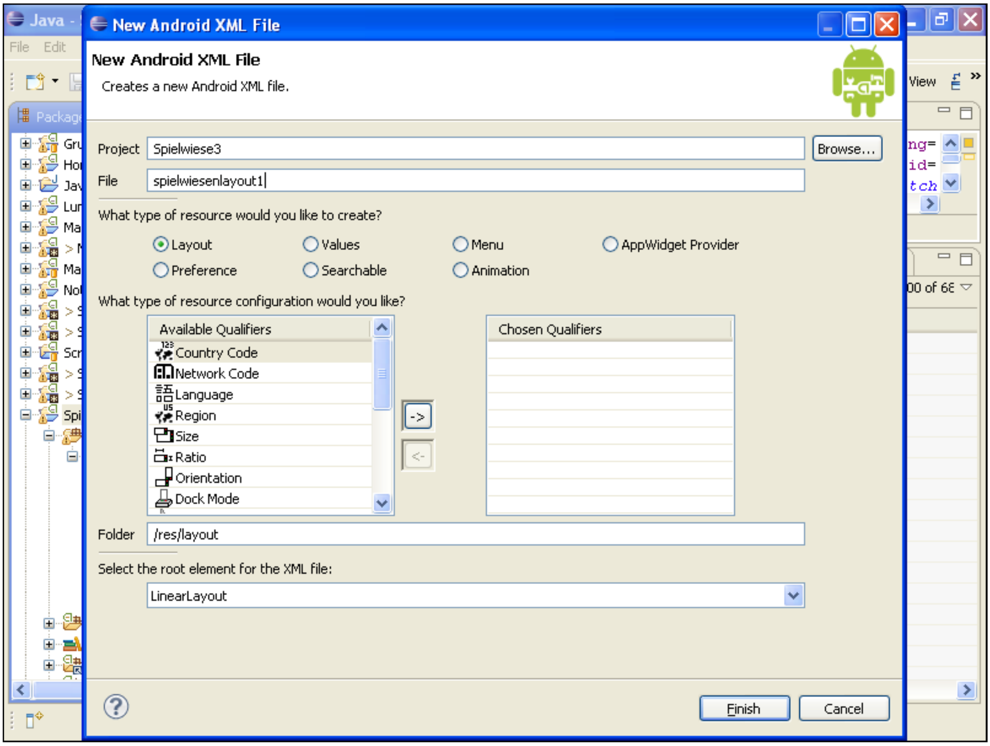


Abbildung 3.12: **Der Android XML File-Wizard**

Wir erkennen in diesem Dialog die meisten Ressourcen-Typen, die wir bereits besprochen haben. Bemerkenswert ist der mittlere Teil des Dialogs, der uns bei der konfigurationsabhängigen Anlage von Ressourcen unterstütz. Hier können wir auswählen, von welcher Konfiguration unsere Ressource abhängig ist (Language, Orientation, ...). Der Wizard kümmert sich automatisch darum, auch die richtige Reihenfolge der Abhängigkeiten einzuhalten.

Wir befinden uns im Projekt Spielwiese3 und müssen nun einen Dateinamen für die Ressource festlegen.

ACHTUNG

Wir müssen immer daran denken, dass die Dateien komplett kleingeschrieben werden und einen gültigen Java-Bezeichner ergeben müssen, da einige Dateinamen zu den Ressourcen-IDs umgewandelt werden, die wir z.B. über `R.layout.spielwiesenlayout1` zum Zugriff auf die Ressourcen benutzen.

Wir wählen hier als Ressourcen-Typ das Layout. Dadurch wird bereits der Folder vorbelegt, und auch das Root-Element des XML-Files wird mit einer sinnvollen Vorgabe belegt, im Falle der Layouts können wir hier nämlich auswählen, in welchen Layout-Typ wir unsere Widgets einbringen wollen:

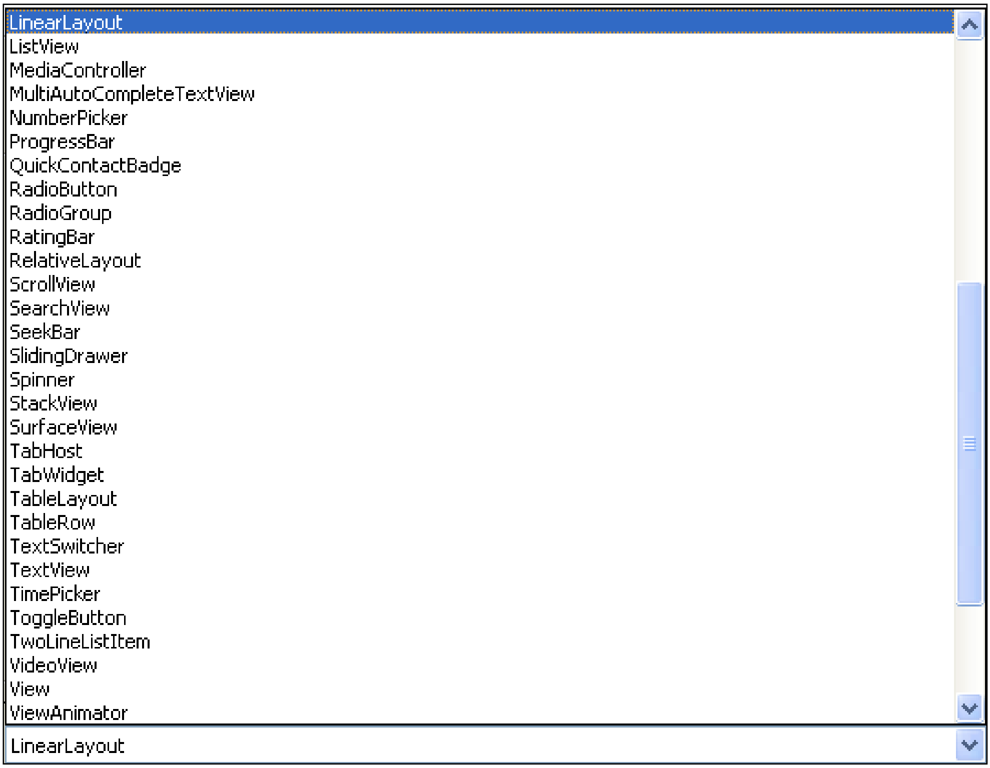


Abbildung 3.13: **Ein ganzer Strauß von Layouts und Widgets**

Das ist auch ein guter Zeitpunkt, um ein bisschen detaillierter auf die Layouts einzugehen. Android bietet eine ganze Menge an nützlichen Layouts und Widgets, und wir wollen uns hier einige davon anschauen.

Aber erstellen wir erst einmal durch einen Klick auf *Finish* unser Layout und öffnen den Layout-Editor. Wie die meisten Helferlein des ADT-Plug-ins bietet der Layout-Editor zwei Sichten an. Zum einen den eigentlichen Editor, mit dem wir interaktiv die Benutzeroberfläche aufbauen können und zum anderen die Sicht auf die XML-Datei. Manchmal kann es nützlich oder schneller, sein direkt in der XML-Datei zu arbeiten.

Durch den Wizard wurde die XML-Datei `spielwiesenlayout1.xml` in unserem `res\layout`-Verzeichnis angelegt. Wenn wir konfigurationsabhängige Parameter ausgewählt hätten, hätte der Wizard auch die jeweiligen Verzeichnisse automatisch erstellt.

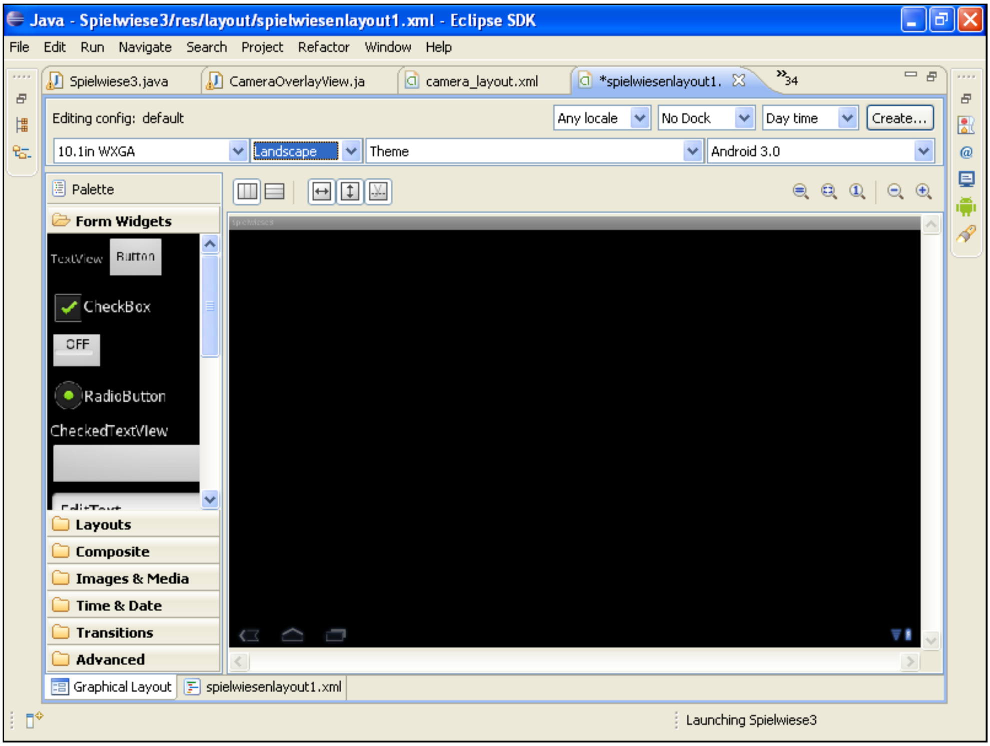


Abbildung 3.14: **Der Layout-Designer**

Nach dem Start des Layout-Designers sollten wir zuerst links oben die Größe des Bildschirms auswählen, denn der Editor startet immer mit dem kleinsten Bildschirm. Da wir uns mit Android 3 beschäftigen, habe ich die 10.1-Zoll-WXGA-Variante gewählt.

Diese Einstellung hat aber keinen Einfluss auf unsere XML-Datei, sondern nur auf die Voransicht. Wir sollten sowieso immer darauf achten, möglichst von der Bildschirmgröße unabhängige Layouts zu gestalten, um so viele Geräte wie möglich zu adressieren.

Das Verhalten auf unterschiedlichen Bildschirmen können wir hier schon vorab testen, in dem wir immer mal wieder die Ansicht umschalten.

Genauso verhält es sich mit der Orientierung (Landscape oder Portrait), dem Theme und der Zielplattform. Die Themes hängen wiederum auch von der Zielplattform ab, so bietet Android 3 mit dem Theme `Theme.Holo.Light` ein Theme an, das auf niedrigeren Leveln nicht vorhanden ist.

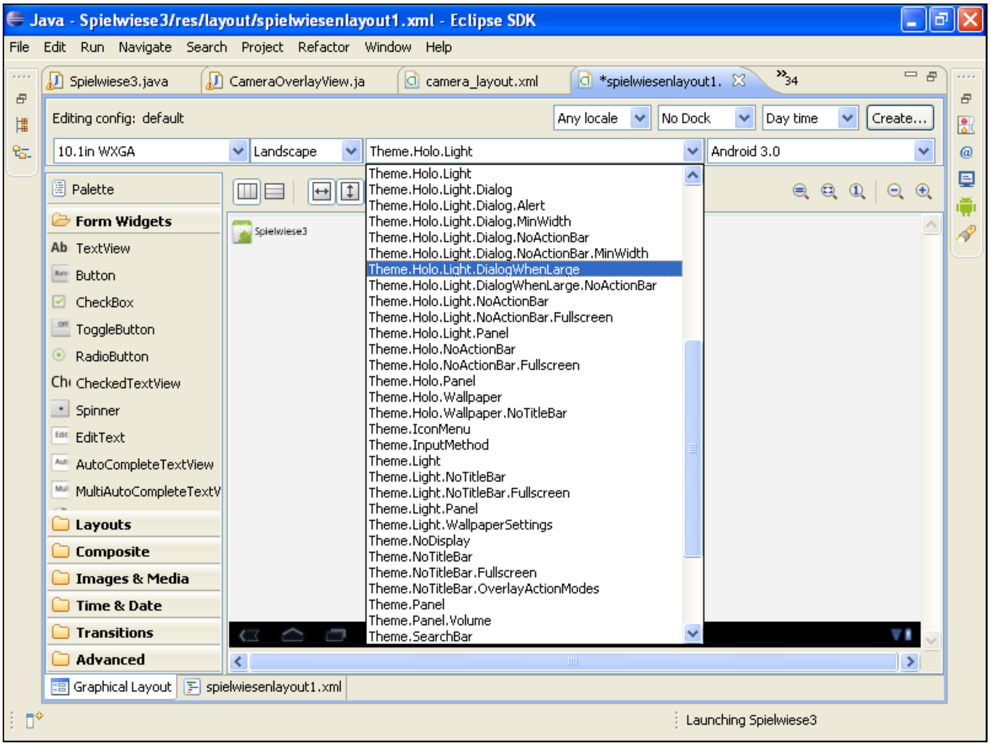


Abbildung 3.15: **Auswahl der Themes**

Der Layout-Editor ist also auch eine gute Anlaufstelle, um die vorhandenen Themes anzuschauen und vorab auszuprobieren. Je nach gewähltem Theme werden auch die Widgets entsprechend im Layout-Editor dargestellt.

Unsere XML-Datei sieht bis jetzt noch sehr übersichtlich aus:

Listing 3.17: **Layout spielwiesenview1.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent" android:orientation="vertical">
</LinearLayout>
```

Die Layouts kann man nachträglich natürlich bearbeiten, in dem man einfach die XML-Datei im Package-Explorer wieder öffnet.

Wir haben hier das `LinearLayout` gewählt, und dieses Layout ist wohl eins der am häufigsten verwendeten Layouts. Das `LinearLayout` ordnet die zugefügten Widgets entweder von oben nach unten oder von links nach rechts an.

Jetzt wollen wir kurz die wichtigsten Layouts durchgehen.

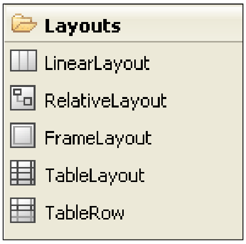


Abbildung 3.16: **Layouts im Layouteditor**

LinearLayout

Die wichtigsten Attribute des `LinearLayout` sind:

```
android:layout_width="match_parent"
android:layout_height="match_parent"
android:orientation="vertical"
```

Mit `Orientation` geben wir an, ob die Elemente horizontal oder vertikal angeordnet werden.

Jedem Element **in** einem `LinearLayout` können dann noch die Attribute

```
android:layout_weight="<ganze zahl>"
```

und

```
android:layout_gravity="top | bottom | left | right | center_vertical |
fill_vertical | center_horizontal | fill_horizontal | center | fill | clip_
vertical | clip_horizontal"
```

mitgegeben werden.

Mittels `android:layout_weight` können wir den Anteil des Elements am zur Verfügung stehenden Platz bestimmen. Ist das Attribut nicht angegeben, so nutzt das Widget keinen verfügbaren Platz.

Mit `android:layout_gravity` wird bestimmt, wo sich das Element innerhalb des Layouts in seinem umgebenden Container ausrichtet.

FrameLayout

Das `FrameLayout` ist das einfachste Layout-Objekt. Alle hinzugefügten Views werden einfach übereinandergestapelt und in der linken oberen Ecke festgepinnt. Das heißt, dass immer nur die zuletzt hinzugefügte View sichtbar ist, es sei denn, diese neue View hat einen transparenten Hintergrund oder ist kleiner als die darunter liegende View.

Das Framelayout kann man ganz gut für Überlagerungstechniken verwenden, z.B. um über einer (darunterliegenden) View Steuerelemente einzublenden.

Listing 3.18: **Beispiel für FrameLayout und RelativeLayout**

```
<?xml version="1.0" encoding="utf-8"?>
    <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <view class="de.androidpraxis.SystenAndHardwareLibrary.CameraView"
    android:id="@+id/camera_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    </view>
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <view class="de.androidpraxis.SystenAndHardwareLibrary.CameraOver➥
            layView"
         android:id="@+id/camera_overlay_view"
         android:layout_width="match_parent"
        android:layout_height="match_parent">
        </view>
        <view class="de.androidpraxis.SystenAndHardwareLibrary.CompassView"
        android:id="@+id/compass_view"
        android:layout_centerHorizontal="true"
        android:layout_alignParentBottom="true"
        android:layout_width="match_parent"
        android:layout_height="75dp"
        android:layout_marginBottom="3dp">
        >
        </view>
    </RelativeLayout>
</FrameLayout>
```

Hier wird das `FrameLayout` benutzt, um die `CameraView` mit einem `RelativeLayout` zu überdecken. In dem `RelativeLayout` ist eine `CameraOverlay` eingebettet, über die noch durch eine relative Positionierung eine `CompassView` gelegt wird.

In der `CameraOverlay`-View sollen später noch Aktionen auf dem Live-Kamerabild oder dem Schnappschuss durchgeführt werden können.

RelativeLayout

Das `RelativeLayout` bietet uns die Möglichkeit, komplexere Layouts zu gestalten, bei denen sich die Ausrichtung von Elementen auf die Ausrichtung anderer Elemente bezieht.

Die enthaltenen Elemente können mit den folgenden Attributen ausgerichtet werden.

ACHTUNG

Diese Attribute werden nicht auf dem `RelativeLayout` definiert, sondern auf den direkt enthaltenen Views.

android:layout_above="@id/<ID>"	Positioniert die untere Kante über der referenzierten View. Stellt diese View oberhalb der referenzierten View dar.
android:layout_alignBaseline="@id/<ID>"	Positioniert die Basislinie auf der Basislinie der referenzierten View.
android:layout_alignBottom="@id/<ID>"	Positioniert die untere Kante an der unteren Kante der referenzierten View, ggf. noch in Verbindung mit android:layout_alignRightOf=»@id/<ID>«.
android:layout_alignLeft="@id/<ID>"	Positioniert die linke Kante an der linken Kante der referenzierten View.
android:layout_alignParentBottom="true"	Richtet die untere Kante der View an der unteren Kante des Containers aus.
android:layout_alignParentLeft="true"	Richtet die linke Kante der View an der linken Kante des Containers aus.
android:layout_alignParentRight="true"	Richtet die rechte Kante der View an der rechten Kante des Containers aus.
android:layout_alignParentTop="true"	Richtet die obere Kante an der oberen Kante des Containers aus..
android:layout_alignRight="@id/<ID>"	Richtet die rechte Kante der View an der rechten Kante der referenzierten View aus.
android:layout_alignTop="@id/<ID>"	Richtet die obere Kante der View an der oberen Kante der referenzierten View aus.
android:layout_alignWithParentIfMissing="true"	Kann die referenzierte View aus layout_alignTop etc. nicht gefunden werden, dann wird der Container als Referenz benutzt.
android:layout_below="@id/<ID>"	Richtet die obere Kante der View unterhalb der unteren Kante der referenzierten View aus. Stellt diese View unterhalb der referenzierten View dar.
android:layout_centerHorizontal="true"	Richtet die View horizontal mittig im Container aus.
android:layout_centerInParent="true"	Zentriert die View im Container.
android:layout_centerVertical="true"	Zentriert die View vertikal im Container.
android:layout_toLeftOf="@id/<ID>"	Richtet die rechte Kante der View an der linken Kante der referenzierten View aus. Stellt diese View links von der referenzierten View dar.
android:layout_toRightOf="@id/<ID>"	Richtet die linke Kante der View an der rechten Kante der referenzierten View aus. Stellt diese View rechts von der referenzierten View dar.

Tabelle 3.14: **Übersicht über die Layout-Parameter des RelativeLayout**

Um zwischen den Elementen noch Abstände zu erzeugen bietet sich entweder das Attribut `android:padding` an oder die Verwendung von Rändern mittels `ansroid:layout_margin<Left|Right|Top|Bottom>`.

TableLayout und TableRow

`TableLayout` und `TableRow` dienen des Layouts in tabellarischer Form.

Listing 3.19: **Beispiel für ein TableLayout**

```
<?xml version="1.0" encoding="utf-8"?>
    <TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1">
    <TableRow>
        <TextView
        android:text="@string/table_layout_4_open"
        android:padding="3dip" />
        <TextView
        android:text="@string/table_layout_4_open_shortcut"
        android:gravity="right"
        android:padding="3dip" />
    </TableRow>
    <TableRow>
        <TextView
        android:text="@string/table_layout_4_save"
        android:padding="3dip" />
        <TextView
        android:text="@string/table_layout_4_save_shortcut"
        android:gravity="right"
        android:padding="3dip" />
    </TableRow>
</TableLayout>
```

Das Beispiel stellt eine Menüstruktur mit Shortcuts nach. Die Shortcuts sind mit `android:gravity="right"` in der 2. Spalte rechts angeordnet.

ACHTUNG

Man kann in einem `TableLayout` keine Zellen andere Zellen überspannen lassen, wie das in HTML mit dem `cellspan`-Attribut möglich ist. Wenn eine Tabelle maximal zwei Spalten hat, dann hat jede Zeile zwei Spalten, auch wenn eine Zeile weniger Spalten beinhalten könnte.

Kleine Zwischenbilanz

Unsere Benutzeroberfläche startet also immer mit einem Layout, das dann die Widgets und ggf. weitere Layout-Elemente enthält. Womit wir dieses Layout nun füllen können, betrachten wir im nächsten Abschnitt.

3.11.4 Füllen des Layouts mit Widgets und anderem

Das Layout alleine macht noch keinen Sommer. Wir benötigen Widgets für Textdarstellung, Texteingabe, wollen Bilder darstellen oder irgendwelche Daten in Listenform.

Der Layout-Editor liefert all diese Komponenten mit.



Abbildung 3.17: **Paletten mit Widgets 1**



Abbildung 3.18: **Paletten mit Widgets 2**

Wir füllen nun unser Layout indem wir ein Widget aus einer Palette auf die Bearbeitungsfläche ziehen.

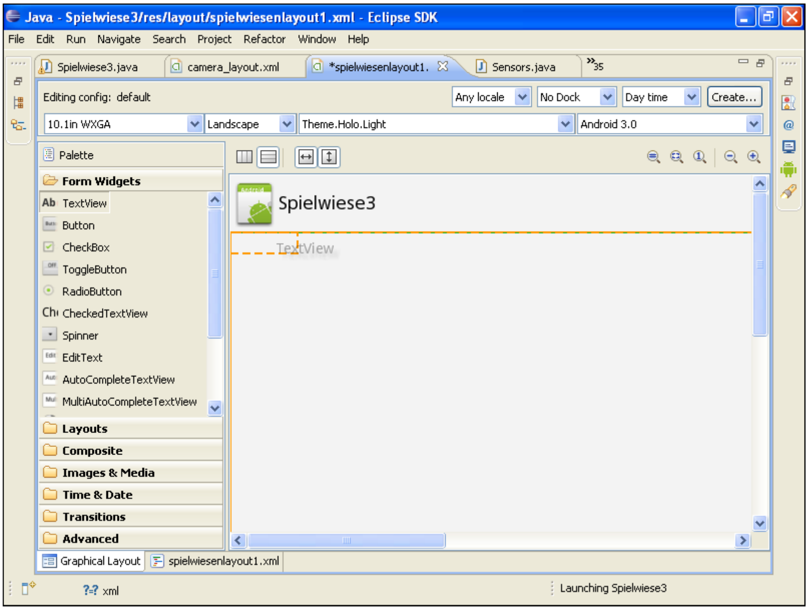


Abbildung 3.19: **Ziehen einer TextView auf das Layout**

Und jetzt geht's im Prinzip erst richtig los. Nun müssen wir die Eigenschaften einstellen, eine ID vergeben, den Text bearbeiten und das Aussehen des Textes verändern. Dabei werden wir sehen, wie uns das Tool dabei unterstützt, und wo wir ggf. noch manuell nacharbeiten müssen.

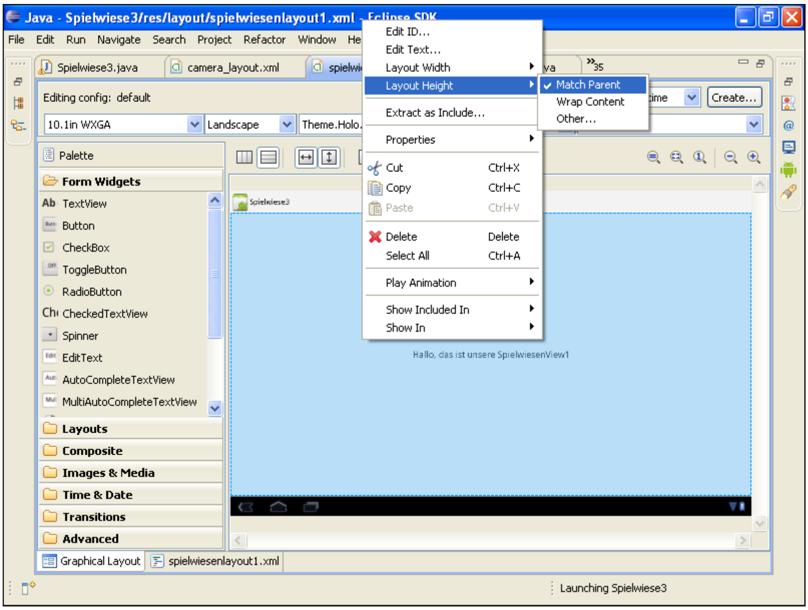


Abbildung 3.20: **Einstellen der Höhe und Breite auf Match Parent**

Die Eigenschaften lassen sich über das Kontextmenü ganz gut bearbeiten. Die wichtigsten Eigenschaften sind hier im oberen Bereich zu erkennen, weitere Eigenschaften erhält man über das *Properties* Flyout-Menü.

Über *Edit ID* können wir die ID des Elements festlegen. Indem Dialog müssen wir lediglich den Namen der ID angeben, **kein** @id: bzw. @+id:, das erledigt das Werkzeug automatisch.

Wir sollten uns angewöhnen, für die meisten Widgets IDs zu vergeben da, wir in der Regel im Programm nochmal auf die Widgets zugreifen müssen.

Hier wollen wir noch den Text festlegen. Das ist eine gute Übung um die Integration des Ressourcensystems auszuprobieren.

Über *Edit Text...* gelangen wir zum *Resource Chooser*.

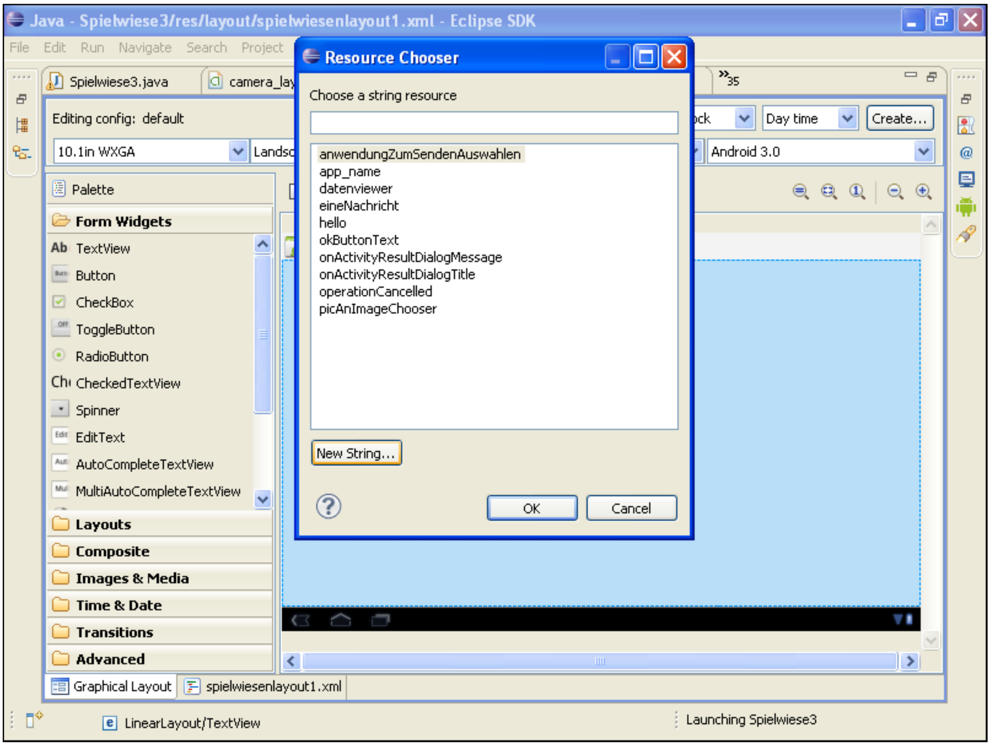


Abbildung 3.21: **Der Resource Chooser**

Hier können wir aus den bestehenden String-Ressourcen auswählen oder aber, und das ist gut, eine neue String-Ressource anlegen.

Mit *New String...* gelangen wir in den Editor für String-Ressourcen, der uns ebenfalls wie der Android XML-File Wizard einige Hilfsmittel zum Erstellen konfigurationsabhängiger Strings an die Hand gibt.

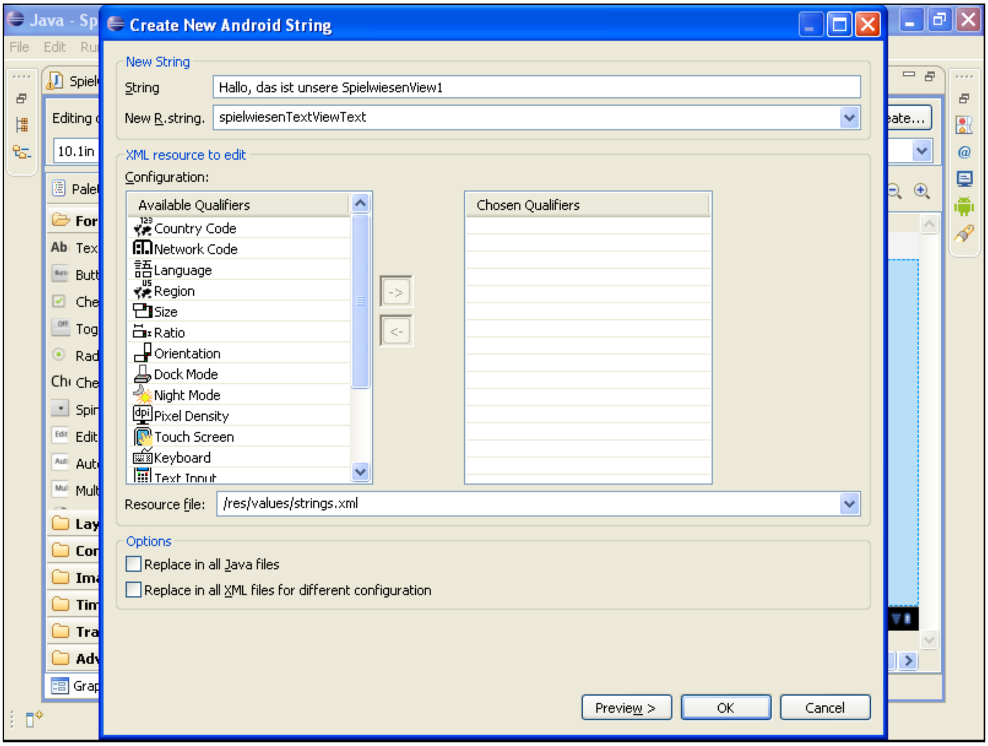


Abbildung 3.22: **Der String-Resurceneditor**

Hier wollen wir allerdings den Default-String erfassen, also den, der nicht konfigurationsabhängig ist bzw. dann genommen wird, wenn kein konfigurationsabhängiger String gefunden werden konnte.

Im Eingabefeld *String* erfassen wir den Text, in *New R.string.* geben wir den Identifier ein, unter dem wir den String dann aus den Ressourcen oder der Anwendung heraus ansprechen.

Mit *OK* übernehmen wir die neue Zeichenkette und bestätigen den nächsten *Resource Chooser* auch mit *OK*.

Unsere XML-Datei sieht dann wie folgt aus:

Listing 3.20: **Layout der SpielwiesenView1**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
android:orientation="vertical">
<TextView
    android:textAppearance="?android:textAppearanceLarge"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent" android:gravity="center"
    android:id="@+id/dieTextView"
    android:text="@string/spielwiesenTextViewText">
</TextView>
</LinearLayout>
```

Man beachte das Attribut `android:textAppearance`. Das Attribut summiert Größe, Stil und Farbe des Textes, und wir benutzen hier eine Referenz auf den Stil `?android:textAppearance-Large` des ausgewählten Themes.

Hinter dieser Stilreferenz findet sich der Stil `TextAppearance.Large`, der wie folgt definiert ist:

Listing 3.21: **Stilreferenz TextAppearance.Large**

```
<style name="TextAppearance.Large">
    <item name="android:textSize">22sp</item>
    <item name="android:textStyle">normal</item>
    <item name="android:textColor">?textColorPrimary</item>
</style>
```

Aus welchem Theme nun die Referenz angewendet wird, legen wir mittels des android:theme-Attributes im Manifest entweder für unsere ganze Anwendung fest oder pro deklarierter Activity:

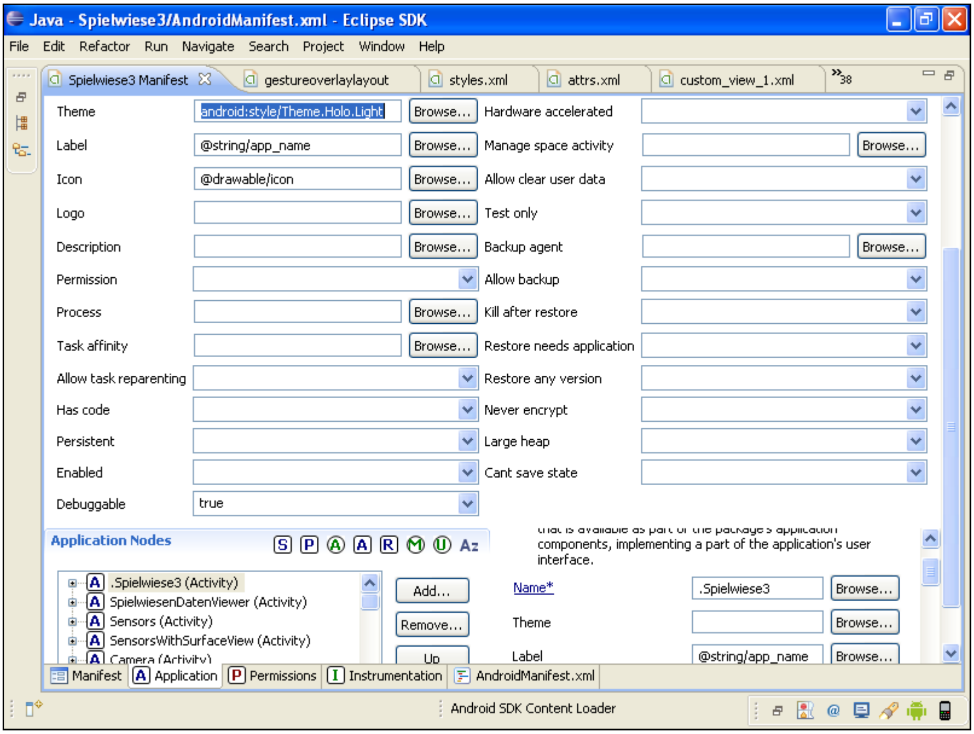


Abbildung 3.23: **Auswahl des Themes für die Anwendung**

Mit der Verwendung der Theme-Referenz können wir das Look and Feel unserer Anwendung anpassen und komplett individualisieren. Geben wir **kein** Theme an, verwendet unsere Anwendung das Standardtheme der Plattform, auf der wir die Anwendung laufen lassen. Der Stil z.B. für großen Text ist entsprechend auch für andere Plattformen und Themes definiert. Da wir auch eigene Themes und Styles erstellen bzw. bestehende Themes erweitern können, ist es möglich, unsere Anwendung komplett zu individualisieren. Allerdings sollte das nur sehr sparsam eingesetzt werden, denn im Grunde erwartet der Anwender durchaus eine gewisse Konsistenz einer Benutzeroberfläche über alle seine Anwendungen hinweg.

TIPP

Die vordefinierten Android-Styles und Themes finden wir hier:

`http://developer.android.com/guide/topics/ui/themes.html`

Ganz am Ende der Seite findet sich ein Link auf die XML-Dateien im aktuellen Repository der aktuellen Android-Version.

Damit hätten wir unser Layout definiert, ein zentrierter großer Text in einem `TextView` in einem `LinearLayout`.

Wie wir in der Palettenübersicht schon sehen, gibt es einige interessante Widgets und zusammengesetzte Elemente, von denen wir uns die wichtigsten noch genauer ansehen werden. Sehr interessant sind z.B. die TabHosts und TabWidgets, aber auch die ListView, GaleryView, ImageView und die Buttons sind natürlich betrachtenswerte Elemente.

Vorher wollen wir uns aber noch um ein paar Benutzeroberflächenelemente kümmern, die nichts mit unserer View direkt, aber doch etwas mit unserer Anwendung zu tun haben. Wir benötigen ja auch Menüs, Dialoge und Benachrichtigungen, und wir wollen auch auf Benutzereingaben reagieren.

3.11.5 Menüs und die Action Bar

Menüs, klar, legen wir auch über den Android-XML Wizard an.

TIPP

Falls einmal nach dem Erstellen der XML-Datei nicht der entsprechende Editor aufgerufen wird, sondern einfach ein XML-Editor der Eclipse, dann können wir einfach die Datei schließen und neu öffnen. Dann klappt es meistens mit dem Editor.

Es gibt drei Arten von Menüs, die sich dadurch unterscheiden, wo sie erscheinen.

1. Optionenmenü
2. Kontextmenü
3. Untermenü

Das Optionenmenü dient den allgemeinen Auswahlmöglichkeiten einer Activity, z.B. dem Aufruf von Einstellungen, dem Anlegen von neuen Daten, also alle Aktionen, die unsere Activity anbietet und nicht mit irgendeiner Auswahl auf dem Bildschirm zu tun haben.

Das Kontextmenü wiederum ist (eigentlich) abhängig davon was gerade auf dem Bildschirm ausgewählt wurde. Zwar kann man Kontextmenüs generell für alle Arten von Views erlauben und auch öffnen, wenn im Prinzip nichts ausgewählt ist, aber eigentlich sollte mit dem Kontextmenü immer eine aktuelle Auswahl behandelt werden, z.B. das Öffnen des aktuellen angetippten Datensatzes in einer Liste.

Untermenüs wiederum treten entweder in Optionenmenüs oder in Kontextmenüs auf, wenn ein Menüpunkt angetippt wird, der wiederum weitere Menüpunkte enthält. Untermenüs können aber keine Icons beinhalten, und sie können auch keine weiteren Untermenüs beinhalten.

Listing 3.22: **Beispielmenü**

```
<?xml version="1.0" encoding="utf-8"?>
<menu
  xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:title="@string/menueItemOption1" android:id="@+id/option1"
android:icon="@drawable/icon"></item>
    <item android:title="@string/menuItemOption2" android:id="@+id/opti➥
        on2"></item>
    <item android:id="@+id/option3" android:title="@string/menuItemOption3">
    <menu>
        <item android:id="@+id/option31" android:title="@string/menuItemOp➥
            tion31"></item>
        <item android:id="@+id/option32" android:title="@string/menuItemOp➥
            tion32"></item>
    </menu>
    </item>
</menu>
```

Die Definition von Optionenmenüs und Kontextmenüs ist gleich. Lediglich wo wir die Menüs dann unserer Anwendung zur Verfügung stellen, unterscheidet sich:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
}
```

In dieser Methode können wir das Kontextmenü für die View `v` erstellen, wobei die View in `menuInfo` weitere Informationen mitschickt. Im Fall einer `ListView` werden Informationen über den Adapter (den Datenlieferanten) mitgeschickt. Damit unsere View ein Kontextmenü anfordert, müssen wir die View mit `registerForContextMenu(<view>)` in der Activity registrieren.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    return super.onCreateOptionsMenu(menu);
}
```

In dieser Methode können wir das Optionenmenü erstellen. Bei beiden Arten können wir die Menüs entweder programmtechnisch aufbauen oder aus einer Menu-Ressource laden.

Das Laden aus der Ressource geht mit einem sogenannten MenuInflater, der aus den Ressourcen-XML-Dateien die Objekte erzeugen kann.

Listing 3.23: **Erstellen eines Menüs aus der Menü-Ressource und Reaktion auf den angetippten (angeklickten) Menüeintrag**

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
            inflater.inflate(R.menu.optionsmenue, menu);

    return super.onCreateOptionsMenu(menu);
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    Toast.makeText(getApplicationContext(), item.getTitle(), Toast.LENGTH_
SHORT).show();
    switch (item.getItemId())
    {
        case R.id.option1:
        break;
        case R.id.option2:
        break;
        case R.id.option31:
        break;
        case R.id.option32:
        break;
    }
    return super.onOptionsItemSelected(item);
}
```

Listing 3.24: **Erstellen eines Kontextmenüs und Reaktion auf ausgewählte Menüeinträge**

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    registerForContextMenu(getListView());
}
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getMenuInflater();
            inflater.inflate(R.menu.optionsmenue, menu);
}
@Override
public boolean onContextItemSelected(MenuItem item) {
    Toast.makeText(getApplicationContext(), "Contextmenu "+item.getTitle(),
Toast.LENGTH_SHORT).show();
    switch (item.getItemId())
    {
    …
    }
    return super.onContextItemSelected(item);
}
```

Die IDs der Menüeinträge sind die IDs, die wir in der XML-Datei angegeben haben.

Manchmal ist es notwendig, die Menüeinträge zu verändern, bevor das Menü erscheint, um ggf. Optionen zu sperren, die momentan nicht verfügbar sind, oder den Check-Status von Einträgen zu setzen.

Dazu überschreiben wir die Methode `onPrepareOptionsMenu(Menu menu)`, in der wir diese Aktionen durchführen können.

ACHTUNG

Vor Android 3 wurde diese Methode immer aufgerufen, bevor das Menü geöffnet wurde, also immer dann wenn der Anwender das Menü angefordert hat. Ab Android 3 können aber Menüeinträge des Optionenmenüs in der Action Bar als sogenannte Action Items ständig sichtbar sein. Daher wird die Methode unter Android 3 nicht regelmäßig aufgerufen. Wir müssen als Entwickler daran denken, immer dann `invalidateOptionsMenu()` aufzurufen, wenn wir Änderungen am Menü in `onPrepareOptionsMenu()` durchführen wollen.

Um Einträge als *Action Items* anzulegen, müssen wir in der Menüdefinition lediglich das Attribut `android:showAsAction="ifRoom|withText"` einfügen.

Action Items sollten die Aktionen sein, die im aktuellen Kontext der Aktivität am häufigsten genutzt werden; in einem E-Mail-Programm z.B. die Knöpfe für »neue Nachricht« oder »Nachricht beantworten«.

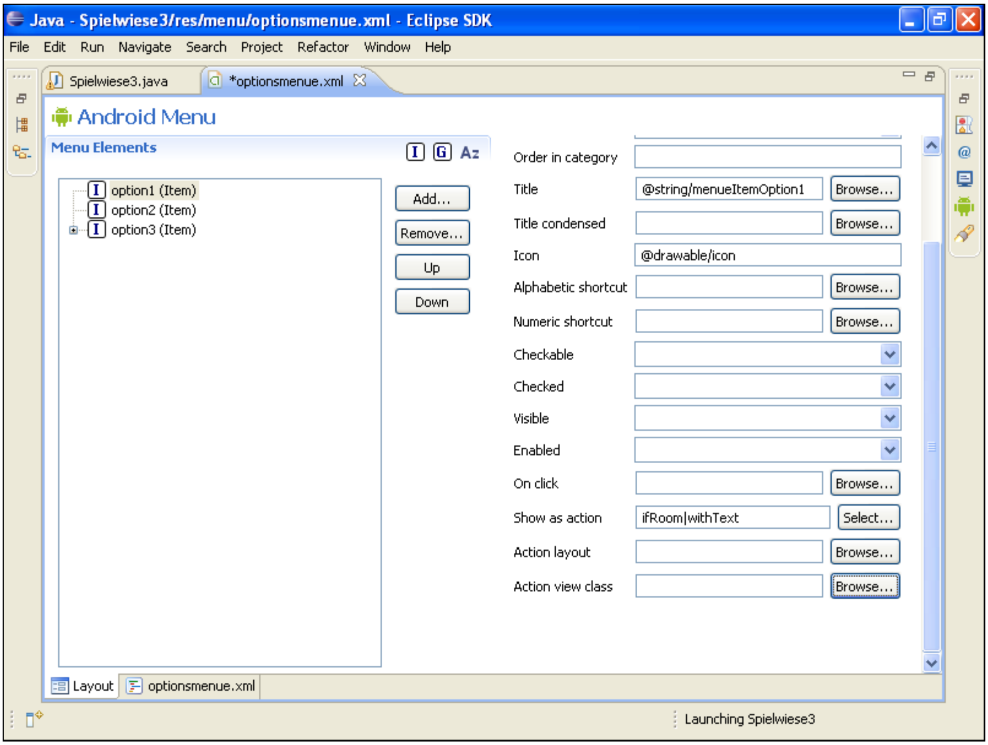


Abbildung 3.24: **Menüeinträge im Editor bearbeiten**

3.11.6 Auf Benutzereingaben reagieren

Einige Views implementieren die Reaktion auf Benutzereingaben bereits perfekt. Selbstverständlich müssen wir uns bei einem TextEdit nicht mehr um die Texteingabe und das Scrollen im Text kümmern, und auch die ListView oder SrollView übernimmt die ihr zugedachten Aufgaben.

Wenn wir aber unser eigenes Malprogramm erstellen oder in einer eigenen View auf Touch-Events, Tastendrücke oder auch auf Menü-Ereignisse reagieren wollen, müssen wir uns anschauen, wie das Framework diese Ereignisse verarbeitet.

Wie wir bereits in den Grundlagen besprochen haben arbeiten die modernen Benutzeroberflächen ereignisorientiert. Tastendrücke, die Interaktion mit dem Touchscreen und andere Hardwarekomponenten lösen Ereignisse aus, die durch den Kernel zum Framework und dort in die Ereignisbearbeitung geschickt werden.

Wir können auf diese Ereignisse mit zwei Mechanismen reagieren, je nachdem, wo wir auf Tastendrücke etc. reagieren wollen:

1. Mittels Event-Listener
2. Durch Überschreiben von Event-Handlern

Event-Listener sind Klassen in unserer Anwendung, die eine entsprechende Schnittstelle implementieren, um auf ein Ereignis reagieren zu können. Event-Handler sind Methoden, die wir in abgeleiteten Klassen überschreiben können. Im Prinzip werden diese Handler von vordefinierten Listenern angesprochen, die Handler dienen mithin als »Abkürzung« zur Ereignisbearbeitung.

Die Komponenten, mit denen der Benutzer direkt interagiert sind die Views. Aus diesem Grund werden die wichtigen Event-Handler dort bereits definiert:

void onCreateContextMenu (ContextMenu menu)	Wird aufgerufen, wenn ein Kontextmenü für die View erstellt werden soll. Hier können wir spezielle Menüeinträge für die View dem Kontextmenü hinzufügen.
boolean onDragEvent(DragEvent event)	Wird aufgerufen, wenn in der View etwas per Drag&Drop-Mechanismen bewegt wird. Drag&Drop-Operationen werden mittels startDrag(...) ausgelöst.
boolean onFilterTouchEvent-ForSecurity(MotionEvent event)	Ähnlich wie normale Touch-Events, wird jedoch aufgerufen, wenn Touch-Events bei teilweise überdecktem Fenster herausgefiltert werden sollen. Über setFilterTouchesWhenObscured(true) werden Touch-Events verworfen, falls die View durch Toasts (Meldungsfesnter) oder andere Fenster teilweise verdeckt wird, um unbeabsichtigte Aktionen zu verhindern.

Tabelle 3.15: **Event-Handler in Views**

void onFocusChanged(boolean gainFocus, int direction, Rect previouslyFocusedRect)	Wird aufgerufen, wenn die View den Fokus erhält oder verliert.
boolean onKeyDown(int keyCode, KeyEvent event)	Wird aufgerufen, wenn eine Taste gedrückt wird. Dazu muss die View den Eingabefokus erhalten können, das muss mittels setFocusable(true) gesetzt werden. Tastaturereignisse werden sowohl von der Hardwaretastatur (sofern vorhanden), einem D-Pad als auch von der Softwaretastatur erzeugt. Manche Trackball-Ereignisse werden ggf. ebenfalls in Tastaturereignisse umgewandelt, um z.B. ein D-Pad zu simulieren.
boolean onKeyLongPress(int keyCode, KeyEvent event)	Wird aufgerufen, wenn eine Taste lange gedrückt wird. Um diese Nachricht überhaupt zu erhalten, muss in onKeyDown auf dem Ereignis die Methode startTracking() aufgerufen werden.
boolean onKeyMultiple(int keyCode, int repeatCount, KeyEvent event)	Wird aufgerufen, wenn es zu einer Taste mehrere Varianten gibt (z.B. Umlaute oder Smileys ...)
boolean onKeyPreIme(int keyCode, KeyEvent event)	Wird aufgerufen bevor der IME (InputMethod-Editor) den Tastendruck empfängt. Hier könnte z.B. die BACK-Taste abgefangen werden bevor die virtuelle Tastatur die Taste empfängt und sich schließt.
boolean onKeyShortcut (int keyCode, KeyEvent event)	Wird aufgerufen, wenn ein Tastenkürzel-Ereignis erkannt wird. Tastenkürzel (Shortcuts) werden durch die Kombination aus der ALT-Taste und einer Zahl oder einem Buchstaben erzeugt.
boolean onKeyUp(int keyCode, KeyEvent event)	Wird aufgerufen, wenn eine Taste losgelassen wird. Die Standard-Implementierung dieses Handlers erzeugt ein abgeleitetes onClick()-Event, falls die Enter-Taste der Tastatur oder die Mitteltaste des D-Pads gedrückt wird.
protected void onOverScrolled(int scrollX, int scrollY, boolean clampedX, boolean clampedY)	Wird aufgerufen, wenn eine Scroll-Operation über die natürlichen Grenzen abgeschlossen wurde. Scrolling wird allerdings nicht automatisch durchgeführt, das muss mit den entsprechenden scrollBy(...)- oder scrollTo(...)-Aufrufen und weiteren Methoden selbst realisiert werden.
void onScrollChanged(int l, int t, int oldl, int oldt)	Wird als Reaktion auf einen Scroll-Vorgang aufgerufen. Scrolling wird allerdings nicht automatisch durchgeführt, das muss mit den entsprechenden scrollBy(...)- oder scrollTo(...)-Aufrufen und weiteren Methoden selbst realisiert werden.

Tabelle 3.15: **Event-Handler in Views (Forts.)**

boolean onTouchEvent (MotionEvent event)	Wird aufgerufen, wenn eine Interaktion mit dem Touchscreen stattfindet.
boolean onTrackballEvent (MotionEvent event)	Wird aufgerufen, wenn ein Trackball benutzt wird.

Tabelle 3.15: **Event-Handler in Views (Forts.)**

Einige Event-Handler liefern einen booleschen Wert zurück. Wenn wir die Event-Handler überschreiben und das Ereignis selbst behandelt haben, signalisieren wir mit der Rückgabe von `true`, dass das Ereignis nicht weiter behandelt werden soll. Andernfalls wird das Ereignis an weitere Handler oder Listener übergeben. Mit der Rückgabe von true oder false bestimmen wir also den weiteren Fluss des Ereignisses. Wie wir im Folgenden sehen werden, können neben den Event-Handlern der View auch Event-Handler der umschließenden Activity, aber auch Event-Listener dazu eingesetzt werden die Ereignisse an anderer Stelle zu verarbeiten.

TIPP

Die Entscheidung, wo ein Ereignis verarbeitet wird, hängt damit zusammen, wo wir die entsprechende Funktionalität am besten realisieren. Die Reaktion auf Touch-Events sowie auf Tastatureingaben oder Steuerungsereignisse per D-Pad oder Trackball behandeln wir, da diese Ereignisse direkt unsere View betreffen, in der Regel direkt auf der View und nicht auf der Activity. Übergeordnete Ereignisse wie Menü-Ereignisse würde ich in der Regel aber auf der umschließenden Activity behandeln, da Menüereignisse oft in die übergreifende Logik der Activity eingebettet sind.

Es fällt vielleicht auf, dass es wenig Ereignisse gibt, die in irgendeiner Form mit dem Scrollen von Views zu tun haben. Die Basisklasse View liefert lediglich rudimentäre Funktionen für das Implementieren von Scrolling, die Ausgestaltung müssen wir je nach dem selbst auf Basis der oben genannten Event-Handler vornehmen. Für vertikales oder horizontales Scrolling gibt es glücklicherweise die Klassen `ScrollView` und `HorizontalScrollView`, bemerkenswerterweise gibt es aber keine Klasse, die beides gleichzeitig bietet. Die `WebView` kann das, implementiert das aber selbst.

Aber wenden wir uns erst einmal wieder der grundlegenden Ereignisbehandlung zu. Neben der View implementieren die Activities ebenfalls vordefinierte Event-Handler, die wir zur Reaktion auf Benutzereingaben überschreiben können. Die wichtigsten Handler sind diejenigen, die die Menüereignisse behandeln. Tastaturereignisse, Touch-Events etc., die von keiner View innerhalb der Activity behandelt wurden, können hier ebenso behandelt werden.

INFO

Wenn eine View ein entsprechendes Ereignis behandelt hat, dann wird dieses nicht mehr an die Activity übergeben. Man kann aber die dispatch*Event(...)-Handler überschreiben um Ereignisse vor der Übergabe an die Views auf der Activity abzuhandeln.**

void onContextItemSelected (MenuItem)	Wird aufgerufen, wenn ein Eintrag aus dem Kontextmenü ausgewählt wurde. Abgeleitete Klassen wandeln häufig den Kontext noch adäquat um und rufen einen eigenen Handler auf, der z.B. im Falle einer ListActivity die Auswahl einer Liste beinhaltet.
void onContextMenuClosed(Menu)	Das Kontextmenü wurde geschlossen.
void onCreateContextMenu(Context Menu, View, ContextMenuInfo)	Das Kontextmenü wird erstellt. Dieser Handler wird **immer** aufgerufen, wenn das Menü angezeigt wird. Damit kann man die Menüeinträge auch abhängig vom Kontext erstellen und z.B. Einträge ausblenden oder deaktivieren, die momentan keine sinnvolle Funktion haben.
void onCreateOptionsMenu(Menu)	Wird aufgerufen, wenn das Optionenmenü der Anwendung erstellt wird. Dieser Handler wird nur **einmal** aufgerufen. Wenn Einträge des Optionenmenüs ausgeblendet oder deaktiviert werden sollen, bietet sich der Handler onPrepareOptionsMenu(...) an.
boolean onKeyDown(int, KeyEvent)	Ähnlich der Handler auf eine View, allerdings werden diese Handler nur aufgerufen, wenn **keine** View das Ereignis selbst behandelt hat.
boolean onKeyLongPress(int, KeyEvent)	
boolean onKeyMultiple(int, int, KeyEvent)	
boolean onKeyShortcut(int, KeyEvent)	
boolean onKeyUp(int, KeyEvent)	
boolean onOptionsItemSelected (MenuItem)	Wird aufgerufen, wenn ein Eintrag des Optionenmenüs ausgewählt wurde.
void onOptionsMenuClosed(Menu)	Wird aufgerufen, wenn das Optionenmenü geschlossen wurde.
boolean onPrepareOptionsMenu (Menu menu)	Wird aufgerufen **bevor** das Optionenmenü angezeigt wird.
boolean onTouchEvent(MotionEvent)	Wird aufgerufen, wenn ein Touch-Event auftritt.
boolean onTrackballEvent (MotionEvent)	Wird aufgerufen, wenn ein Trackball-Event auftritt.

Tabelle 3.16: **Event-Handler auf der Activity**

void onUserInteraction()	Wird aufgerufen, wenn irgendein Benutzerereignis erzeugt wird. Dieses Ereignis ist dazu gedacht, die Aktionen auszuführen, die nötig sind wenn der Benutzer mit der Activity in irgendeiner Form interagiert. Es können dann z.B. ausstehende Benachrichtigungen der Activity gelöscht werden.
void onUserLeaveHint()	Wird aufgerufen, wenn der Benutzer die Activity »verlässt«, in der Regel vor onPause(), aber der Aufruf ist nicht garantiert.
boolean dispatchKeyEvent(KeyEvent)	Ähnlich der Event-Handler, diese Handler werden aber aufgerufen, **bevor** das Ereignis an die Views übergeben wird. Damit könnte man die Ereignisse abfangen und behandeln bevor irgendeine View das erledigt.
boolean dispatchKeyShortcutEvent (KeyEvent)	
boolean dispatchTouchEvent (MotionEvent)	
boolean dispatchTrackballEvent (MotionEvent)	

Tabelle 3.16: **Event-Handler auf der Activity (Forts.)**

Fragmente führen einige wenige Handler ein, die sich lediglich mit der Behandlung von Menüereignissen befassen.

boolean onContextItemSelected(MenuItem)	Diese Handler arbeiten wie die Handler auf der Activity.
Boolean onCreateContextMenu (ContextMenu, View, ContextMenuInfo)	
boolean onCreateOptionsMenu (Menu, MenuInflater)	
boolean onOptionsItemSelected(MenuItem)	
boolean onOptionsMenuClosed(Menu)	

Tabelle 3.17: **Event-Handler auf dem Fragment**

Neben den Event-Handlern können wir andere Klassen als Listener bei den Views anmelden. Damit ist es möglich, bestimmte Ereignisse an anderer Stelle, z.B. in der Activity oder einem Fragment abzuhandeln, das diese View beinhaltet, bzw. wir können auf der View auch bereits abgeleitete Ereignisse abhandeln.

View.OnClickListener	void onClick(View)
View.OnCreateContextMenuListener	onCreateContextMenu(ContextMenu, View, ContextMenuInfo)

Tabelle 3.18: **Listener Interfaces für das Horchen auf Ereignisse der View**

View.OnDragListener	onDrag(View, DragEvent)
View.OnKeyListener	onKey(View, int, KeyEvent)
View.OnLongClickListener	onLongClick(View)
View.OnTouchListener	onTouch(View, MotionEvent)

Tabelle 3.18: **Listener Interfaces für das Horchen auf Ereignisse der View (Forts.)**

Wir können einen Listener per `View.setOn****Listener(…)` bei der View anmelden. Wie wir sehen, gibt es viel mehr Listener als Event-Handler auf der View vorbereitet sind. Das hängt damit zusammen dass einige der Ereignisse, die durch die Event-Handler abgedeckt werden können, bereits abgeleitete Ereignisse sind. Diese werden aus den grundlegenden Ereignissen quasi berechnet. Ein Click-Event muss z.B. aus einem Touch-Event und aus Tastatur-Events abgeleitet werden, ebenso ist ein LongClick-Event ein Click-Event, das für eine gewisse längere Zeit anliegt.

Dieses Konzept abgeleiteter Ereignisse wird später noch weiter getrieben, sodass aus einer Serie von Touch-Events z.B. Gesten wie die beliebte Schleudergeste (Fling) oder die Zoom-Geste erzeugt werden, auf die man wiederum horchen kann.

TIPP

Durch das Setzen des OnClick()-Listeners wird die View direkt "clickable", d.h. wir müssen das nicht unbedingt explizit durch das Attribut android:clickable="true" bzw. setClickable(true) anzeigen. Ebenso verhält es sich mit dem OnLongClick()-Listener und dem Attribut android:longClickable="true" bzw. setLongClickable(true). Wenn wir allerdings lediglich ein Kontextmenü einer View zuordnen, das auf den Long-Click geöffnet werden soll, dann müssen wir das Attribut explizit setzen. Implizit passiert das wiederum, wenn auf der Activity registerForContextMenu(...) bzw. setOnCreateContextMenuListener(...) aufgerufen wird.

Bevor man nun Ereignisse verarbeiten kann, ist es wichtig zu wissen, unter welchen Umständen bestimmte Ereignisse überhaupt an die View und/oder die Activity weitergeleitet werden.

Um Tastaturereignisse zu empfangen muss eine View überhaupt den Eingabefokus erhalten dürfen. Das erreichen wir durch das Attribut `android:focusable="true"` oder durch den Aufruf `setFocusable(true)`. Dadurch werden Views so konfiguriert, dass Sie den Eingabefokus durch Navigieren per D-Pad, Trackball oder Tastatur mittels Cursortasten erhalten und in Folge dann auch Tastaturereignisse empfangen können.

Wichtig ist aber auch, den Unterschied zwischen dem Toch-Mode und dem normalen Eingabemodus zu verstehen. Wenn wir ein Gerät mit Touchscreen haben, dann wählen wir in der Regel ein Element per »Fingerzeig« aus. In diesem Moment befindet sich das Element im Touch-Mode. Bei einem Button soll dann direkt das Click-Event erzeugt werden, das ist auch das normale Verhalten, aber er sollte nicht den Eingabefokus erhalten. Den Eingabefokus soll ein Button nur durch Navigation per Cursortasten oder Trackball erhalten können. Das Click-Event wird dann durch Betätigen der Enter-Taste oder der Trackball-Taste ausgelöst.

Haben wir allerdings ein Eingabefeld, dann soll dieses durch den »Fingerzeig« sehr wohl den Eingabefokus erhalten. Die View `EditText` regelt das bereits, wenn wir eine eigene View haben, die sich wie ein Eingabefeld verhalten soll, dann müssen wir diese mit dem Attribut `android:focusableInTouchMode="true"` bzw. `setFocusableInTouchMode(true)` entsprechend konfigurieren. Erst dann erhalten wir in der View auch Tastaturereignisse und können auch erst dann z.B. die virtuelle Tastatur für die View anzeigen.

Listing 3.25: **Konfigurieren der eigenen View, um Tastaturereignisse zu empfangen**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<LinearLayout
     xmlns:android="http://schemas.android.com/apk/res/android"
     android:orientation="horizontal"
     android:layout_width="match_parent"
     android:layout_height="wrap_content">
    <Button android:layout_width="wrap_content" android:layout_height="wrap_
content" android:id="@+id/button1" android:text="Button 1"/>
    <Button android:layout_width="wrap_content" android:layout_height="wrap_
content" android:id="@+id/button2" android:text="Button 2"/>
</LinearLayout>
<de.androidpraxis.SpielwieseLibrary3.EventHandlerView
android:id="@+id/eventhandlerview"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:longClickable="true"
android:hapticFeedbackEnabled="true"
android:focusable="true"
android:focusableInTouchMode="true"

android:soundEffectsEnabled="true"
android:inputType="text"
/>
</LinearLayout>
```

Mit obiger Konfiguration ist es dann möglich, in der View `EventHandlerView` die Tastaturereignisse zu empfangen:

Listing 3.26: **Event-Handler für Tastaturereignisse**

```
public class EventHandlerView extends View implements View.OnClickListener {
[…]
private void init()
{
    setOnClickListener(this);
}
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    Log.d(Globals.LOG_TAG,"Event: onKeyDown(int keyCode, KeyEvent event)");
    return super.onKeyDown(keyCode, event);
}
```

```
@Override
public boolean onKeyUp(int keyCode, KeyEvent event) {
    Log.d(Globals.LOG_TAG,"Event: onKeyUp(int keyCode, KeyEvent event)");
    return super.onKeyUp(keyCode, event);
}
@Override
public void onClick(View v) {

    Log.d(Globals.LOG_TAG,"Listener: onClick(View view)");
    InputMethodManager imgr = (InputMethodManager)getContext().
getSystemService(Context.INPUT_METHOD_SERVICE);
    imgr.showSoftInput(this, InputMethodManager.SHOW_FORCED);
}

[…]
}
```

In der Spielwiese wird nun für diese View die virtuelle Tastatur angezeigt, wenn die View angeklickt wird. Hierfür benötigen wir dann einen Listener, mit dem wir das Anklicken abfangen können.

Diesen Listener können wir entweder auf der View selbst oder aber auf der Activity realisieren. Hier ist es sinnvoll, das Verhalten auf der View zu realisieren, denn die View entscheidet ja darüber, dass eine virtuelle Tastatur angezeigt werden soll.

ACHTUNG

Die Methoden um Listener anzumelden, lauten `setOn***Listener(…)`. Das impliziert dass genau ein Listener gesetzt werden kann, ansonsten lauten die Methoden in der Regel `addOn***Listener(…)` oder ähnlich. Wenn wir also in der View den onClick()-Listener setzen und die umschließende Activity setzt ebenfalls den Listener, dann würde dieser neue Listener den alten ersetzen.

Click-Events sind hauptsächlich bei Buttons von Interesse, um auf das Drücken des Buttons zu reagieren. Da ein Button selbst eine View ist, kann man auf jedem Button einen OnClick-Listener registrieren, entweder jeweils einen eigenen Listener pro Button oder einen gemeinsamen Listener, der alle Buttons behandelt. In der Regel werden wir einen gemeinsamen Listener registrieren, der den Button anhand seiner ID identifiziert. Der Vorteil ist, dass dann weniger Klasseninstanzen erzeugt werden müssen und der Code für die Ereignisbehandlung an zentraler Stelle übersichtlich zusammengehalten wird. Buttons lösen ähnlich wie Menüereignisse eine gewisse Applikationslogik aus, die zwar mit der View zusammenhängt, aber doch eher im Kontext der Activity oder des umgebenden Fragments sinnvoll umzusetzen ist, denken wir z.B. an den Button »Senden« oder »Speichern« oder Ähnliches. Daher definieren wir die onClick()-Listener für Buttons in der Regel innerhalb der Activity oder des Fragments und setzen dann für die Buttons diesen gemeinsamen onClick()-Listener.

Listing 3.27: **OnClick()-Listener für Buttons**

```
public class EventHandlerActivity extends Activity implements OnClickListe➥
    ner,
OnCreateContextMenuListener,
OnDragListener,
```

```
OnFocusChangeListener,
OnKeyListener,
OnLongClickListener,
OnTouchListener {
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.eventhandlerlayout);
    View view = findViewById(R.id.eventhandlerview);
    if (view!=null)
    {
        registerForContextMenu(view);
        view.setOnDragListener(this);
        view.setOnFocusChangeListener(this);
        view.setOnKeyListener(this);
        view.setOnLongClickListener(this);
        view.setOnTouchListener(this);

        InputMethodManager imgr = (InputMethodManager)
getSystemService(Context.INPUT_METHOD_SERVICE);
        imgr.showSoftInput(view, 0);
    }

    view = findViewById(R.id.button1);
    if (view!=null)
    {
        view.setOnClickListener(this);
    }
    view = findViewById(R.id.button2);
    if (view!=null)
    {
        view.setOnClickListener(this);
    }

}
[…]
@Override
public void onClick(View view) {
    Log.d(Globals.LOG_TAG,"Event in Activity - Listener: onClick(View
view)");
    if (view instanceof Button)
    {
        Button btn = (Button)view;
        Toast.makeText(this, "onClick(): "+btn.getText(), Toast.LENGTH_
LONG).show();
        switch (btn.getId())
        {
        case R.id.button1:
        //Aktion für Button 1
        break;
        case R.id.button2:
        //Aktion für Button 2
        break;
        }
    }
}
[…]
}
```

Im obigen Beispiel implementiert die Activity den OnClick()-Listener und weist diesen den Buttons zu. Innerhalb des Listeners reagieren wir dann anhand der Item-ID auf den jeweils gedrückten Button.

Diese Vorgehensweise ist ähnlich wie bei der Behandlung von Menüereignissen, außer dass wir für Menüereignisse nicht auf den Klick eines Eintrags reagieren sondern einen entsprechenden Event-Handler überschreiben.

Es bietet sich an, wie schon beschrieben, für die Behandlung der Menüereignisse ebenfalls die Activity oder das Fragment zu benutzen und nicht die View selbst auf die Ereignisse reagieren zu lassen.

Die Menüs erstellen wir in den entsprechenden Event-Handlern:

Listing 3.28: **Erzeugen der Menüs**

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo
menuInfo) {
    Log.d(Globals.LOG_TAG,"Event in Activity: onCreateContextMenu(ContextMenu
menu, View v, ContextMenuInfo menuInfo)");
    super.onCreateContextMenu(menu, v, menuInfo);
    getMenuInflater().inflate(R.menu.contextmenu, menu);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    Log.d(Globals.LOG_TAG,"Event in Activity: onCreateOptionsMenu(ContextMenu
menu)");
    //super.onCreateOptionsMenu(menu);
    getMenuInflater().inflate(R.menu.optionsmenu, menu);
    return true;
}
```

Damit die Activity für die entsprechenden Views ein Kontextmenü erzeugen kann muss die Activity entsprechend »angemeldet« werden:

Listing 3.29: **Die Activity bei der View als Kontextmenü-Handler anmelden**

```
View view = findViewById(R.id.eventhandlerview);
if (view!=null)
{
    registerForContextMenu(view);
}
```

Dadurch wird die View automatisch für den LongClick-Handler eingerichtet, sodass bei einem langen Klick das Kontextmenü erstellt und angezeigt wird.

Listing 3.30: **Behandeln der Menü-Ereignisse**

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    Log.d(Globals.LOG_TAG,"Event in Activity: onContextItemSelected(MenuItem
item)");
```

```
    switch (item.getItemId())
    {
    case R.id.item1:
        Toast.makeText(this, "Context Item 1", Toast.LENGTH_SHORT).show();
        return true;
    case R.id.item2:
        Toast.makeText(this, "Context Item 2", Toast.LENGTH_SHORT).show();
        return true;
    }
    return super.onContextItemSelected(item);
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    Log.d(Globals.LOG_TAG,"Event in Activity: onOptionsItemSelected(MenuItem
item)");
    switch (item.getItemId())
    {
    case R.id.option1:
        Toast.makeText(this, "Option Item 1", Toast.LENGTH_SHORT).show();
        return true;
    case R.id.option2:
        Toast.makeText(this, "Option Item 2", Toast.LENGTH_SHORT).show();
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Wichtig ist noch der Handler `onPrepareOptionsMenu(…)`. Da `onCreateOptionsMenu(…)` nur einmal aufgerufen wird, kann man hier keine Einträge abhängig von irgendwelchen Kriterien deaktivieren oder ausblenden. Das kann man in `onPrepareOptionsMenu(…)` erledigen.

Listing 3.31: **Dynamisches Ändern von Menüeinträgen**

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
    Log.d(Globals.LOG_TAG,"Event in Activity: onPrepareOptionsMenu(Menu
menu)");
    MenuItem item = menu.findItem(R.id.item3);
    if (item!=null)
    {
        item.setEnabled(false);
    }
    return super.onPrepareOptionsMenu(menu);
}
```

Neben dem Deaktivieren/Aktivieren könnte zu diesem Zeitpunkt auch der Text oder das Icon geändert werden.

ACHTUNG

Problematisch wird es, wenn die Einträge als Optionen in der Action Bar angezeigt werden. Dann gibt es nämlich, außer beim ersten Mal, für diese Optionen keinen definierten Zeiptunkt zu dem `onPrepareOptionsMenu(…)` aufgerufen werden würde, bzw. es wäre nicht sicher dass diese Einträge rechtzeitig aktualisiert werden, denn sie sind ja ständig sichtbar. Um trotzdem Menüeinträge im Optionenmenü, die in der Action Bar angezeigt werden,

ACHTUNG

dynamisch verändern zu können müssen wir die Methode `invalidateOptionsMenu()` immer dann aufrufen, wenn sich der Zustand unseres Benutzerinterface ändern könnte, also faktisch nach jeder Aktion, die wir ausführen. Wenn es sich dabei allerdings um Dinge handelt, die in hohem Maße vom Kontext der aktuellen View abhängen wäre das Kontextmenü auch der bessere Platz für diese Items.

Jetzt haben wir uns die wichtigsten Ereignisse angeschaut, mit denen wir umgehen müssen. Das sind nun in der Hauptsache Menüeinträge und die Reaktion auf das Anklicken von Knöpfen. Abgeleitete Views, wie z.B. die AdapterViews, führen noch weitere Handler ein, die im Allgemeinen aber von diesen bekannten Handlern abgeleitet sind.

Spannend wird es, wenn wir uns mit den Touch-Events beschäftigen, denn gerade die fingerbasierten Touchscreens mit der Möglichkeit, mehrere Berührungspunkte zu registrieren, bieten per Gesten ganz ausgefallene Möglichkeiten der Interaktion. Gestensteuerung kam schon vor einigen Jahren in Form von Mausgesten auf, und auch optische Gestensteuerung per Kamera ist Gegenstand der Forschung und Entwicklung. Durch die direkte Interaktion mit der Bildschirmoberfläche mit Multitouch-Erkennung fand dann auf die portablen Geräte diese Form der Steuerung große Verbreitung, und sie macht auch großen Spaß; viel mehr Spaß als das ständige Mausgeschubse.

Basis der Gestensteuerung ist das Touch-Event. Das Touch-Event wird über den Handler `onTouchEvent(…)` an die View bzw. Activity übergeben. Da die Touch-Events sehr eng mit der View verknüpft sind, sollten wir die Auswertung auch auf der View erledigen.

Zentrales Element aller Touch- und Gesten-Ereignisse ist das MotionEvent. Das MotionEvent transportiert sämtliche Bewegungsereignisse, egal ob es sich dabei um Finger-/Stiftbedienung, Trackball, Joystick, Maus oder D-Pad handelt. Um aus einer Sequenz von MotionEvents eine entsprechende Geste zu extrahieren, berechnet das Framework mit entsprechenden Algorithmen die jeweilige Geste. So wird ein »Tap« auf den Bildschirm erst als solcher erkannt, wenn eine gewisse kurze Zeit ein Druck auf dem Bildschirm anliegt, so dass versehentliche »Taps« herausgefiltert werden können.

Ein MotionEvent transportiert möglicherweise mehrere Pointer (Zeiger). Das ist z.B. bei der Bedienung mit Fingern der Fall, wenn mehrere Finger benutzt werden (Multitouch). Für jeden Pointer transportiert das Event die aktuellen Positionsdaten, aber auch weitere Daten wie den Druck oder die Größe des Pointers. Ob ein Pointer diese Werte liefern kann, hängt natürlich vom Eingabegerät ab. Ein Touchscreen, der rein kapazitiv arbeitet, kann z.B. den Druck nicht übermitteln, die Größe des Pointers hingegen schon. Resistive Touchscreens können, da hier bauartbedingt sowieso zwei Schichten aufeinandergedrückt werden, möglicherweise den Druck liefern der auf die Schichten ausgeübt wird. Das Framework substituiert allerdings auch Werte anhand anderer Kriterien, so dass eine Art Druck auch bei Touchscreens ausgeworfen wird, die eigentlich gar nicht druckempfindlich sind.

Betrachten wir uns die Auswertung der Ereignisse innerhalb von `onTouchEvent(…)`:

Listing 3.32: **Auswerten der MotionEvents in onTouchEvent**

```
@Override
public boolean onTouchEvent(MotionEvent event) {
if (event.getAction()==MotionEvent.ACTION_DOWN)
{
    penStrokes.clear();
}
if (event.getAction()==MotionEvent.ACTION_MOVE)
{
    final int historySize = event.getHistorySize();
    final int pointerCount = event.getPointerCount();
    for (int h = 0; h<historySize; h++)
    {
        for (int i=0; i<pointerCount; i++)
        {
         int pointerId = event.getPointerId(i);
         Pen pen = new Pen();
         pen.x = event.getHistoricalX(i,h);
         pen.y = event.getHistoricalY(i,h);
         pen.pressure = event.getHistoricalPressure(i,h);
         pen.size = event.getHistoricalTouchMajor(i,h);
         penStrokes.add(pointerId,pen);
         Log.d(Globals.LOG_TAG,pen.toString());
        }
    }
    for (int i=0; i<pointerCount; i++)
    {
        int pointerId = event.getPointerId(i);
        Pen pen = new Pen();
        pen.x = event.getX(i);
        pen.y = event.getY(i);
        pen.pressure = event.getPressure(i);
        pen.size = event.getTouchMajor(i);
        penStrokes.add(pointerId,pen);
        Log.d(Globals.LOG_TAG,pen.toString());
    }
}
invalidate();
return super.onTouchEvent(event);
}
```

Ganz entscheidend sind die Eigenschaften

```
event.getAction()
final int historySize = event.getHistorySize();
final int pointerCount = event.getPointerCount();
```

des MotionEvents.

Mittels `getAction()` ermitteln wir, was gerade passiert. Es gibt hier die Möglichkeiten:

MotionEvent.ACTION_DOWN	Mindestens ein Pointer wurde gedrückt, eine Geste startet.
MotionEvent.ACTION_MOVE	Die Geste wird mit einem oder mehreren Pointern ausgeführt.
MotionEvent.ACTION_HOVER_MOVE	Eine Bewegung findet statt ohne dass der Pointer gedrückt wurde.
MotionEvent.ACTION_UP	Die Geste endet dadurch das alle Pointer losgelassen wurden.
MotionEvent.ACTION_ OUTSIDE	Die Bewegung findet außerhalb der Grenzen des Elements statt.
MotionEvent.ACTION_CANCEL	Die Geste wurde abgebrochen.
MotionEvent.ACTION_POINTER_ DOWN	Ein Pointer (außer dem primären) wurde gedrückt, z.B. ein zusätzlicher Finger kommt ins Spiel.
MotionEvent.ACTION_POINTER_UP	Ein Pointer (außer dem primären) wurde losgelassen, z.B. ein zusätzlicher Finger nimmt nicht mehr an der Geste teil.
MotionEvent.ACTION_SCROLL	Eine Scrollbewegung wurde erkannt, das ist allerdings kein Touch-Event und daher in diesem Zusammenhang nicht relevant.

Tabelle 3.19: **ACTION-Konstanten für MotionEvent-Actions**

Die Methode `getPointerCount()` liefert die Anzahl der aktiven Zeiger im MotionEvent, und `getHistorySize()` liefert die Anzahl historischer Werte des Events. Das Framework kumuliert innerhalb einer Bewegung (`ACTION_MOVE`) die Pointer-Daten, die über `getHistorical***(pointerIndex,historienIndex)` abgefragt werden können. Zusätzlich transportiert jedes MotionEvent auch die brandaktuellen Pointer-Daten.

In unserem Beispiel sammeln wir die Daten der Pointer und benutzen die Daten zum Zeichnen von Kreisen, deren Größe aus der Größe des Touch-Bereichs und deren Deckkraft (Alpha-Kanal) aus dem Druck ermittelt werden.

In einer konkreten Fingermal-Anwendung müssten wir noch die Empfindlichkeit für Größe und Druck konfigurieren, um z.B. feine Malarbeiten möglich zu machen. Interessant ist hier, dass das System versucht, die tatsächliche durch Stift oder Finger bedeckte Fläche in Gerätepixeln zu ermitteln:

```
event.getTouchMajor(pointerIndex)
event.getTouchMajor(pointerIndex)
event.getToolMajor(pointerIndex)
event.getToolMinor(pointerIndex)
```

Diese Aufrufe liefern die kürzere und die längere Achse der Ellipse, die durch den Stift oder die Finger bedeckt sind, und sind dem Aufruf von `getSize(pointerIndex)` vorzuziehen.

Unser Beispiel ist die Grundlage für weitere Experimente, denn bisher erkennen wir ja keine Gesten, sondern zeichnen einfach die berührten Punkte nach. Es ist aber schon sehr schön zu sehen wie die historischen Daten gesammelt werden um dann etwas daraus zu machen. Genauso arbeiten die Detektoren, die das Framework mitliefert und die wir nutzen können, um bestimmte Gesten zu erkennen und wiederum als Ereignis übermittelt zu bekommen.

Es gibt den vorgefertigten GestureDetector und den ScaleGestureDetector, die jeweils einfache Gesten erkennen und als Ereignisse übermitteln können:

	GestureDetector(Context context, GestureDetector.OnGestureListener listener)	Erstellen des GestureDetectors mit einem zugeordneten Listener. Der Detector kann nur vom UI Thread aus erstellt werden. Wenn ignoreMultitouch auf true gesetzt wird, ignoriert der Detector alle Gesten die mehr als einen Finger beinhalten. Das ist nützlich, wenn der Detector zusammen mit dem ScaleGestureDetector eingesetzt werden soll.
	GestureDetector(Context context, GestureDetector.OnGestureListener listener, Handler handler)	
	GestureDetector(Context context, GestureDetector.OnGestureListener listener, Handler handler, boolean ignoreMultitouch)	
boolean	onTouchEvent(MotionEvent ev)	Damit füttern wir den Detector mit den MotionEvents aus unserer View. Liefert die Methode true zurück, hat der Detector das Event verwendet, ansonsten gehört es wohl nicht zu einer Geste, die der Detector erkennen könnte.
void	setIsLongpressEnabled(boolean isLongpressEnabled)	Wenn LongPressEnabled auf false gesetzt wird, wird die Geste für einen langen Druck nicht umgesetzt. Dann kann der Detector dazu verwendet werden, Scroll-Gesten zu erkennen, die mittels Druck und Bewegen des Fingers ausgelöst werden.
void	setOnDoubleTapListener(GestureDetector. OnDoubleTapListener onDoubleTapListener)	Setzt zusätzlich zum normalen Listener einen Listener an den Doppelberührungen (Doubletaps), also »Doppelklicks« übergeben werden.

Tabelle 3.20: **Methoden des GestureDetectors**

boolean	onDown(MotionEvent e)	Ein Tap wurde erkannt, und das auslösende MotionEvent beim Drücken wird übergeben.
boolean	onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY)	Ein »Schleuderereignis« wurde erkannt. Es werden die Ereignisse beim Drücken und beim Loslassen übergeben sowie die Geschwindigkeit in X- und Y-Richtung.
void	onLongPress(MotionEvent e)	Ein langer Druck wurde erkannt.
boolean	onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY)	Ein Scroll-Ereignis wurde erkannt. **Wichtig:** setLongPressEnabled(false) muss dazu auf dem Detector aufgerufen worden sein.
void	onShowPress(MotionEvent e)	Es wurde ein Druck erkannt, aber der Benutzer hat noch nicht wieder losgelassen.
boolean	onSingleTapUp(MotionEvent e)	Ein einzelner »Tap« wurde erkannt, und das Ereignis beim Loslassen wird übergeben.

Tabelle 3.21: **Listener-Methoden: GestureDetector.OnGestureListener**

boolean	onDoubleTap(MotionEvent e)	Ein doppelter »Tap« wurde erkannt.
boolean	onDoubleTapEvent (MotionEvent e)	Hier werden die Ereignisse übergeben, die während der Erkennung auftreten (in der Regel die Sequenz Down, Up, Down, Up).
boolean	onSingleTapConfirmed (MotionEvent e)	Ein einzelner »Tap« wurde erkannt.

Tabelle 3.22: **Listener Methoden: GestureDetector.OnDoubleTapListener**

Der »Doubletap« wird gerne für ein schnelles Vergrößern einer Ansicht benutzt, z.B. in der Webseitenansicht oder in der Bildgalerie zum Ein- und Auszoomen an dem Punkt, den man antippt, seltener auch als »Doppelclick« um Einträge zu öffnen.

ScaleGestureDetector(Context context, ScaleGestureDetector.OnScaleGestureListener listener)		Erstellen des Detectors mit dem entsprechenden Listener
float	getCurrentSpan()	Aktuelle Distanz zwischen den Fingern, die die Geste formen.

Tabelle 3.23: **Methoden des ScaleGesture-Detectors**

float	getCurrentSpanX()	Aktuelle Distanz in X-Richtung
float	getCurrentSpanY()	Aktuelle Distanz in Y-Richtung
float	getFocusX()	X-Koordinate des Start- bzw. Referenzpunkts. Das ist die Position, an der die Scale-Geste gestartet wurde.
float	getFocusY()	Y-Koordinate des Startpunkts.
float	getPreviousSpan()	Liefert die vorherige Distanz zwischen den Fingern zurück.
float	getPreviousSpanX()	
float	getPreviousSpanY()	
float	getScaleFactor()	Liefert den aktuellen Skalierungsfaktor gegenüber dem vorherigen Scale-Event.
long	getEventTime()	Liefert die aktuelle Zeit, zu der das Ereignis erkannt wurde.
long	getTimeDelta()	Liefert den Zeitunterschied in Millisekunden zum vorherigen erkannten Ereignis.
boolean	isInProgress()	Liefert true falls die Geste noch andauert.
boolean	onTouchEvent(MotionEvent event)	Hiermit füttern wir den Detector mit den MotionEvents. Liefert true falls der Detector das Ereignis verarbeitet hat.

Tabelle 3.23: **Methoden des ScaleGesture-Detectors (Forts.)**

boolean	onScale(ScaleGestureDetector detector)	Wird aufgerufen, während die Scale-Gesture ausgeführt wird.
boolean	onScaleBegin(ScaleGesture-Detector detector)	Wird zu Beginn der Geste aufgerufen.
void	onScaleEnd(ScaleGesture-Detector detector)	Wird bei Abschluss der Geste aufgerufen, d.h. wenn die Finger vom Display genommen werden.

Tabelle 3.24: **Listener-Methoden: ScaleGestureDetector.OnScaleGestureListener**

An die Methoden des Listeners wird der Detector übergeben. Wir können dann die aktuellen und auch historischen Werte für die aktuelle Geste aus dem Detector ermitteln und entsprechend auswerten.

Mit diesen einfachen Detectoren ist es bereits möglich, die grundlegenden Gesten für unsere eigene Anwendung zu nutzen. Auf der Spielwiese nutzen wir das dafür, mittels Scale-Gesture einen Kreis zu zeichnen und diesen per Scroll-Gesture frei zu »scrollen«:

Listing 3.33: **Einsatz des OnGestureDetectors und OnScaleGestureDetectors**

```
public class GestureEventView extends View implements OnGestureListener,
OnDoubleTapListener, OnScaleGestureListener {
private Paint paint = new Paint();
private GestureDetector gestureDetector = null;
private ScaleGestureDetector scaleGestureDetector = null;
private float radius = 0.0f;
private float FocusX = 0.0f;
private float FocusY = 0.0f;
public GestureEventView(Context context) {
    super(context);
    init();
}
public GestureEventView(Context context, AttributeSet attrs) {
    super(context, attrs);
    init();
}
public GestureEventView(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
    init();
}
private void init()
{
    gestureDetector = new GestureDetector(getContext(),this,null,true);
    gestureDetector.setOnDoubleTapListener(this);
    gestureDetector.setIsLongpressEnabled(false);
    scaleGestureDetector = new ScaleGestureDetector(getContext(),this);
}
@Override
public boolean onTouchEvent(MotionEvent event) {
    gestureDetector.onTouchEvent(event);
    scaleGestureDetector.onTouchEvent(event);
    return super.onTouchEvent(event);
}
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawCircle(this.FocusX, this.FocusY, this.radius, paint);
}
@Override
public boolean onScale(ScaleGestureDetector detector) {
    Log.d(Globals.LOG_TAG,"onScale(ScaleGestureDetector detector)");
    FocusX = detector.getFocusX();
    FocusY = detector.getFocusY();
    radius = detector.getCurrentSpan()/2.0f;
    invalidate();
return true;
}
@Override
public boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX,
float distanceY) {
    Log.d(Globals.LOG_TAG,"onScroll(MotionEvent e1, MotionEvent e2,
"+distanceX+","+distanceY+")");
    this.FocusX -= distanceX;
    this.FocusY -= distanceY;
    invalidate();
```

```
    return true;
}
[…]
}
```

ACHTUNG

Zwischen onScroll und onFling wird nicht immer »sauber« unterschieden. Nach einem onScroll tritt in den meisten Fällen auch ein onFling auf, und ein onFling ohne onScroll ist so gut wie unmöglich, da die Fling-Geste immer wie eine Scroll-Geste startet. Um die Mehrdeutigkeit zu umgehen, könnte man eine Scroll-Geste erst dann verarbeiten, wenn ein langer Druck erkannt wurde. Oder man muss dafür sorgen, dass sich die Scroll- und die Fling-Geste korrekt ergänzen.

Die nächsthöhere Disziplin ist die Gestensteuerung mittels »frei« definierter Gesten. Dazu bringt das Framework eine komplette Gesten-Verwaltung und eine `GestureOverlayView` mit, mit der Gesten aufgezeichnet und in Verbindung mit einer `GestureLibrary` erkannt werden können. Die Beispielanwendungen des SDK bringen einen rudimentären Editor zum Erstellen von Gesten mit, den man für die Erzeugung von Gesten einsetzen kann.

Egal ob man nun Gesten erzeugt oder Gesten erkennen will, die `GestureOverlayView` ist die Komponente, mit dem wir diese Funktionalität erreichen können. Wie der Name schon sagt, wird diese View **über** andere Views gelegt, so dass die Gestenerkennung im Overlay stattfindet, die darunter liegenden Views ihre eigentliche Funktionalität beibehalten. Die Overlay-View ist zu diesem Zweck transparent gehalten.

Um die View über eine andere View zu legen, benutzen wir das FrameLayout:

Listing 3.34: **Überlagern einer View mit der GestureOverlayView**

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="0dp"
android:layout_weight="1">

    <TextView android:id="@+id/gestureoverlay_textview"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:gravity="center"
    android:text="GestureOverlay"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>

    <android.gesture.GestureOverlayView android:id="@+id/gestureoverlayview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gestureStrokeType="multiple"
    />
</FrameLayout>
```

Da die Views mit dem FrameLayout mit dem ersten Element an unterster Position übereinander gestapelt werden, legen wir das Overlay als letzten und damit obersten Eintrag fest.

Die zugehörige Activity implementiert die benötigten Listener.

Listing 3.35: **Listener auf der GestureOverlayView**

```
public class GestureOverlayActivity extends Activity implements OnGesture➥
    PerformedListener, OnGestureListener, OnGesturingListener {
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
    setContentView(R.layout.gestureoverlaylayout);

    GestureOverlayView gov = (GestureOverlayView)findViewById(R.
id.gestureoverlayview);
    if (gov!=null)
    {
        gov.addOnGestureListener(this);
        gov.addOnGesturePerformedListener(this);
        gov.addOnGesturingListener(this);
    }
}
[…]
@Override
public void onGesturePerformed(GestureOverlayView view, Gesture gesture) {
    Log.d(Globals.LOG_TAG,"onGesturePerformed(GestureOverlayView view, Moti➥
    onEvent event)");
}
}
```

Wirklich relevant ist der `OnGesturePerformed`-Listener, an den eine erkannte Geste übergeben wird. An dieser Stelle können wir nun entweder die Geste anhand einer GestureLibrary erkennen und/oder eine unbekannte Geste in die GestureLibrary einfügen.

Wir können die GestureBuilder-Beispielapplikation aus dem SDK benutzen, um eine Gestenbibliothek anzulegen. Die Anwendung legt die Gesten in der Datei »gestures« unter dem Verzeichnis `Environment.getExternalStorageDirectory()` ab. Diese Datei ist auch von anderen Anwendungen lesbar.

Die Gesten werden unter einem Namen in der Bibliothek abgelegt, und wir können zu einer aufgezeichneten Geste eine Liste der möglichen Gestennamen von der Bibliothek erfragen, wobei die Bibliothek eine Liste möglicher Ergebnisse mit einem Score zurückgibt, der die wahrscheinliche Übereinstimmung mit einer gespeicherten Geste angibt.

Listing 3.36: **Erkennen einer Geste**

```
public class GestureOverlayActivity extends Activity implements OnGesture➥
    PerformedListener, OnGestureListener, OnGesturingListener {

private final File libFile = new File(Environment.getExternalStorageDirecto➥
    ry(), "gestures");
private GestureLibrary library= null;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.gestureoverlaylayout);
```

```
    GestureOverlayView gov = (GestureOverlayView)findViewById(R.
id.gestureoverlayview);
    if (gov!=null)
    {
        gov.addOnGestureListener(this);
        gov.addOnGesturePerformedListener(this);
        gov.addOnGesturingListener(this);
        library = GestureLibraries.fromFile(libFile);
        if (library!=null) library.load();
    }
}
@Override
public void onGesturePerformed(GestureOverlayView view, Gesture gesture) {
    Log.d(Globals.LOG_TAG,"onGesturePerformed(GestureOverlayView view, Moti➥
      onEvent event)");
    Log.d(Globals.LOG_TAG,"Strokes: "+gesture.getStrokesCount());
    recognize(gesture);
}
public void recognize(Gesture gesture)
{
    Prediction candidate = null;
    int strokes = gesture.getStrokesCount();
    ArrayList<Prediction> prediction = library.recognize(gesture);
    if (!prediction.isEmpty())
    {
        candidate = prediction.get(0);
    }
    if (candidate!=null)
    {
        Prediction top = candidate;
        TextView tv = (TextView)findViewById(R.id.gestureoverlay_textview);
        if (tv!=null)
        {
        tv.setText("Gesture detected: "+top.name+" Score: "+top.score);
        }
    }
}
```

Die entscheidenden Aufrufe sind hier

```
library = GestureLibraries.fromFile(libFile);
if (library!=null) library.load();
```

um die Library zu laden und

```
ArrayList<Prediction> prediction = library.recognize(gesture);
```

um die Vermutungen über die Gesten-Kandidaten innerhalb der Library herauszufinden. Über den gefundenen Namen `prediction.name` können wir innerhalb unserer Anwendung dann Aktionen ausführen.

ACHTUNG

Wenn wir Gesten für Löschen oder ähnlich irreversible Aktionen definieren und auswerten, sollten wir immer noch eine Sicherheitsabfrage einbauen. Das macht sich beim Löschen sowieso immer gut, aber gerade wenn das System eine Geste vielleicht mal missdeutet hat, kann das wirklich wichtig zur Schonung der Nerven sein.

Die Klasse `Gesture` liefert uns Zugriff auf einige wichtige Eigenschaften einer Geste, sowohl innerhalb des OnGesturePerformed-Listeners als auch Informationen über die in der Library gespeicherten Gesten. So können wir mittels `gesture.getStrokeCount()` die Anzahl der unabhängigen Striche herausfinden und innerhalb der Kandidaten aus `library.recognize(…)` ggf. noch die heraussuchen, deren Anzahl der Striche mit unserer Geste übereinstimmt. Damit kann man die Genauigkeit ggf. noch erhöhen:

Listing 3.37: **Einbeziehen der Anzahl von Strichen**

```
Prediction candidate = null;
int strokes = gesture.getStrokesCount();
ArrayList<Prediction> prediction = library.recognize(gesture);
for (int i=0; i<prediction.size(); i++)
{
    Log.d(Globals.LOG_TAG,"Prediction: "+prediction.get(i).name+" "+predic➥
      tion.get(i).score);
    ArrayList<Gesture> gs = library.getGestures(prediction.get(i).name);
    for(Gesture g:gs)
    {
        Log.d(Globals.LOG_TAG,"Strokes: "+g.getStrokesCount());
        if (g.getStrokesCount()==strokes)
        {
        if (candidate==null)
        {
        candidate = prediction.get(i);
        break;
        }
        }
    }
}
```

Genauso könnte man noch das umschließende Rechteck einbeziehen, wobei dieses zum Vergleichen der Gesten auf die gespeicherten Gesten normalisiert, also in ein einheitliches Koordinatensystem z.B. der Dimension [(0,0),(1.0,1.0)] überführt werden müsste.

Der Ansatz des GestureBuilders im SDK ist recht simpel. Die Qualität der Gestenerkennung lässt sich noch steigern, indem man einen Trainer baut, mit dem wir für eine benannte Geste unterschiedliche Varianten aufzeichnen und in der Library speichern. `ArrayList<Gesture> gs = library.getGestures(name)` liefert zu einem Namen ja alle zugehörigen Gestenvarianten zurück, die sich mittels `library.addGesture(name,gesture)` hinzufügen lassen.

Mit ein bisschen Geduld lässt sich die Gestensteuerung zu einer einfachen Handschrifterkennung ausbauen, wobei der Anwendungsschwerpunkt wohl eher die Bereitstellung bestimmter Kommandos wie Durchstreichen, Abhaken o.Ä. ist.

In einer Anwendung, um unsere »Reality zu augmentieren«, also unsere Umwelt mit Anmerkungen zu versehen, können wir die Gesten dazu nutzen, um z.B. innerhalb eines Fotos bestimmte Bereiche einzukreisen, um dort zusätzliche Informationen einzugeben.

Bis zu diesem Punkt haben wir uns die Reaktion auf direkte oder indirekte Ereignisse angesehen, und ein Großteil dessen macht die Reaktion auf Tastaturereignisse und Touch-Events aus. Das MotionEvent selbst hat eine noch größere Spannweite und deckt ab Android 3.1 auch Ereignisse von Joysticks, Gamepads und weiteren Eingabegeräten ab, wobei hier der Event-Handler `boolean onGenericMotionEvent(MotionEvent event)` auf den Views bzw. der `OnGenericMotionListener` zum Einsatz kommt. Mit diesem Handler lassen sich alle Ereignisse aller Eingabegeräte verarbeiten.

3.11.7 Eigene Views und Widgets

Da wir Layouts vorzugsweise per XML-Ressourcen anlegen, stellt sich natürlich die Frage, wie wir eigene Widgets erstellen und in Layouts verwenden können.

Wir unterscheiden im Grunde folgende Situationen, in denen wir eigene Widgets oder Views ableiten:

1. Um eine neue View mit neuer Funktionalität zu erstellen
2. Um die Funktionalität einer bestehenden View zu erweitern
3. Um bestehende Widgets in einem neuen Widget zusammenzufassen

Im vorigen Abschnitt haben wir uns eingehend mit Event-Handlern und Listenern beschäftigt, um innerhalb von Views, Widgets und/oder Activities auf Benutzerereignisse zu reagieren. Dabei haben wir schon (fast) eigene Widgets abgeleitet, haben wir doch die Funktionalität einer View durch die speziellen Reaktionen in den Handlern erweitert und auch, z.B. im Falle der `TouchEventView`, auch die Darstellung der View bzw. des Widgets verändert bzw. erweitert.

Neben den Event-Handlern und Listenern ist die Methode `onDraw(Canvas canvas)` der View-Klasse eine entscheidende Stelle, an der wir bestimmen können was unsere View anzeigt.

INFO

Noch einmal ein kurzes Wort zu Views und Widgets. Eine View ist nicht immer automatisch ein Widget, denn Layouts wie das LinearLayout oder das RelativeLayout sind auch Views und stellen selbst nichts dar. Ein Widget wie ein EditText-Widget oder ein Spinner wiederum stellen etwas auf dem Bildschirm dar und bieten auch entsprechende Funktionalität zur Benutzerinteraktion. Widgets sind also in der Regel Dinge, die etwas darstellen und darüber hinaus auch auf Benutzerereignisse reagieren. Die Abgrenzung ist nicht immer scharf, und ein Widget ist immer eine Art View.

Unsere `TouchEventView` ist z.B. eine direkte Ableitung von View und dient dazu, die Touch-Events aufzuzeichnen und als Kreise mit entsprechendem Radius und Deckkraft, abhängig von Druck und Fläche des »Pointers«, anzuzeigen.

Listing 3.38: **Ableitung der View zur Visualisierung der Touch-Events**

```
public class TouchEventView extends View
```

Das ist (fast) alles, was wir zur Erstellung eigener Views und Widgets tun müssen. Allerdings kommt es nun darauf an, was dieses Widget tut, und wir wollen das Widget ja auch ggf. konfigurieren können. Dazu müssen die Konstruktoren überschrieben und modifiziert werden:

Listing 3.39: **Konstruktoren einer View**

```
public TouchEventView(Context context) {
    super(context);
}
public TouchEventView(Context context, AttributeSet attrs) {
    super(context, attrs);
}
public TouchEventView(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
}
```

Wichtig sind die Attribute, die an die Konstruktoren übergeben werden. Wenn wir uns das zugehörige XML-Layout anschauen, finden wir ja immer wieder android:***-Attribute, die wir zur Konfiguration einer View oder eines Widgets benutzen können:

Listing 3.40: **Deklaration der eigenen View in toucheventlayout.xml**

```
<de.androidpraxis.SpielwieseLibrary3.TouchEventView
android:id="@+id/eventhandlerview"
android:layout_width="match_parent"
android:layout_height="0dp"
android:layout_weight="1"

android:clickable="true"
android:longClickable="true"
android:hapticFeedbackEnabled="true"
android:soundEffectsEnabled="true"

/>
```

Hier sehen wir sehr schön, wie wir eigene Views, und auch alle Komponenten, die **nicht** im `android.view`-Package liegen, innerhalb der XML-Dateien deklarieren können. Wir benutzen einfach den voll qualifizierten Klassennamen (`de.androidpraxis.SpielwieseLibrary3.TouchEventView`) als Elementnamen.

Alle android:***-Attribute werden in der Regel durch die Standardimplementierung der View (`super(context, attrs)`) ausgewertet, wir können aber auch selbst eigene Attribute

deklarieren. Diese Attribute müssen aber erst noch als Styleable Resource bekannt gemacht werden und werden in der Regel in der Datei `values\attrs.xml` abgelegt.

Listing 3.41: **Einführen des Attributs penColor für TouchEventView**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="TouchEventView">
        <attr name="penColor" format="color"/>
    </declare-styleable>
</resources>
```

Wir wollen unsere View so konfigurieren können, dass wir die Farbe der Kreise in der XML-Deklaration festlegen können. Dazu legen wir innerhalb der Datei attrs.xml eine »stylable« Ressource für TouchEventView an.

Die eigenen Attribute, die wir deklarieren, können u.a. folgende Formate haben:

color	Farbwert, kann auch eine Referenz auf eine Ressource, z.B. `app:penColor="@color/defaultPenColor"`, oder auf ein Stilattribut, `app:penColor=" ?android:attr/textColorPrimary"`, sein
dimension	Dimension, kann auch eine Referenz auf eine Ressource oder auf ein Stilattribut sein.
boolean	Boolescher Wert, kann auch eine Referenz auf eine boolesche Ressource sein.
string	Eine Zeichenkette, kann auch eine Referenz auf eine String-Ressource sein.
integer	Ganzzahliger Wert, kann auch eine Referenz auf eine Ressource sein.
float	Fließkommazahl, kann auch eine Referenz auf eine Ressource sein
reference	Referenziert eine Stil-Ressource

Tabelle 3.25: **Attributtypen für eigene Attributdeklaration**

Darüber hinaus können wir mit der Attributdefinition Flags (Kombination vorgegebener Werte) und Aufzählungen (ein Wert aus einer Aufzählung vorgegebener Werte) deklarieren:

Listing 3.42: **Aufzählungen und Flags deklarieren**

```
<declare-styleable name="TouchEventView">
    <attr name="penColor" format="color"/>
    <attr name="penShape">
        <enum name="circle" value="0"/>
        <enum name="square" value="1"/>
    </attr>
    <attr name="useValues">
      <flag name="size" value="0x01"/>
      <flag name="pressure" value="0x02"/>
    </attr>
</declare-styleable>
```

In unserem Layout können wir dann die deklarierten Attribute nutzen:

Listing 3.43: **Benutzen eigener Attribute**

```
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res/de.androidpraxis.Spielwiese3"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
<de.androidpraxis.SpielwieseLibrary3.TouchEventView
[…]
        app:penColor="#ff0000"
        app:penShape="circle"
        app:useValues="size|pressure"
/>
</LinearLayout>
```

Wichtig ist die Deklaration des Namensraums unserer Attribute mittels `xmlns:app="http://schemas.android.com/apk/res/de.androidpraxis.Spielwiese3"`. Erst dadurch können wir über `app:penColor="#ff0000"` auf unser Attribut in unserer Anwendung zugreifen. Statt des Präfix app können wir jedes beliebige Präfix verwenden, solange wir es mit dem entsprechenden Namensraum deklarieren.

In unserem Widget verarbeiten wir die eigenen Attribute wie folgt:

Listing 3.44: **Auswerten der eigenen Attribute**

```
public TouchEventView(Context context) {
    super(context);
}
public TouchEventView(Context context, AttributeSet attrs) {
    super(context, attrs);
    initAttributes(context,attrs,0);
}
public TouchEventView(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
    initAttributes(context,attrs,defStyle);
}
private void initAttributes(Context context, AttributeSet attrs,int defSty➥
    le)
{
    TypedArray a = context.obtainStyledAttributes(attrs,R.styleable.
TouchEventView,defStyle,0);
    int penColor = a.getColor(R.styleable.TouchEventView_penColor, 0 );
    penShape = a.getInt(R.styleable.TouchEventView_penShape, SHAPE_CIRCLE);
    useValues = a.getInt(R.styleable.TouchEventView_useValues, FLAG_PRESSURE
| FLAG_SIZE);
    penSize = a.getDimension(R.styleable.TouchEventView_penSize, 25.0f);
    paint.setColor(penColor);
    a.recycle();
}
```

Über `TypedArray a = context.obtainStyledAttributes(attrs,R.styleable.TouchEventView,defStyle,0)` besorgen wir uns ein typisiertes Array, das aus den übergebenen

Attributen in Verbindung mit unserer »styleable« View gebildet wird. Nach diesem Aufruf liegen alle Werte innerhalb des Arrays vor, und wir können dann per `a.get***(R.styleable.TouchEventView_***)` auf unsere jeweiligen Attribute zugreifen. Im Listing ist zu erkennen, dass Flags und Aufzählungen einfach als ganzzahlige Werte übermittelt werden.

Um uns die Verwendung zu erleichtern, benutzen wir innerhalb der Klasse entsprechende Konstanten:

Listing 3.45: **Konstanten für unsere Aufzählung und Flags**

```
public static int SHAPE_CIRCLE = 0;
public static int SHAPE_SQUARE = 1;
public static int FLAG_SIZE = 0x01;
public static int FLAG_PRESSURE = 0x02;
```

Damit wir mit unserer View interagieren können, überschreiben wir den OnTouch-Event-Handler:

Listing 3.46: **Überschreiben des OnTouch(...) Event-Handlers**

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    if (event.getAction()==MotionEvent.ACTION_DOWN)
    {
    penStrokes.clear();
    }
    if (event.getAction()==MotionEvent.ACTION_MOVE)
    {

    final int historySize = event.getHistorySize();
    final int pointerCount = event.getPointerCount();

    for (int h = 0; h<historySize; h++)
    {
        for (int i=0; i<pointerCount; i++)
        {
        int pointerId = event.getPointerId(i);
        Pen pen = new Pen();
        pen.x = event.getHistoricalX(i,h);
        pen.y = event.getHistoricalY(i,h);
        pen.pressure = event.getHistoricalPressure(i,h);
        pen.size = event.getHistoricalTouchMajor(i,h)/2;
        penStrokes.add(pointerId,pen);
        Log.d(Globals.LOG_TAG,pen.toString());
        }
        }

        for (int i=0; i<pointerCount; i++)
        {
        int pointerId = event.getPointerId(i);
        Pen pen = new Pen();
        pen.x = event.getX(i);
        pen.y = event.getY(i);
        pen.pressure = event.getPressure(i);
        pen.size = event.getTouchMajor(i)/2;
        penStrokes.add(pointerId,pen);
```

```
            Log.d(Globals.LOG_TAG,pen.toString());
            }
        }
        invalidate();

        return super.onTouchEvent(event);
}
```

Innerhalb des Event-Handlers werten wir die Touch-Events aus und zeichnen die Punkte auf, die durch die Touch-Events auf dem Bildschirm berührt wurden.

Diese Punkte soll unser eigenes Widget noch darstellen, und für alles, was die Darstellung eines Widgets betrifft, ist die onDraw(Canvas canvas)-Methode verantwortlich:

Listing 3.47: **Überschreiben der onDraw(...)-Methode zum Zeichnen des Widgets**

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    for (int i : penStrokes.getPenIds())
    {
        PenStrokes strokes = penStrokes.getStrokes(i);
        for (Pen pen: strokes.getStrokes())
        {
        if ((useValues & FLAG_PRESSURE) == FLAG_PRESSURE)
        {
        paint.setAlpha((int)(255.0f*pen.pressure));
        }
        float _size = penSize/2.0f;
        if ((useValues & FLAG_SIZE) == FLAG_SIZE)
        {
        _size = pen.size;
        }
        if (penShape == SHAPE_CIRCLE)
        {
        canvas.drawCircle(pen.x,pen.y, _size, paint);
        }
        else
        {
        RectF r = new RectF(pen.x - pen.size, pen.y-pen.size, pen.x + pen.
size, pen.y + pen.size);
        canvas.drawRect(r, paint);
        }
        }
    }
}
```

Wichtig ist dabei immer der Aufruf von `super.onDraw(canvas)`, damit wir die Standardimplementierung zum Zeichnen der View aufrufen. Damit werden z.B. die Hintergründe korrekt gezeichnet, und wir müssen uns darum nicht mehr kümmern. Deshalb sollte in der Regel die Ursprungsmethode als Erstes aufgerufen werden. Wollen wir aber erreichen, dass die Ursprungsmethode unsere Darstellung überlagert, dürfen wir die Methode erst nach dem Zeichnen aufrufen, z.B. wenn wir ein `EditText`-Widget mit Notizblocklinien hin-

terlegen wollen. In diesen Fällen müssen wir aber darauf achten, dass **kein** Hintergrund konfiguriert ist bzw. wir diesen selbst zum richtigen Zeitpunkt zeichnen.

Damit haben wir schon unser eigenes Widget erzeugt, das wir auch in eine Library packen können, um es in mehreren Projekten zu verwenden.

ACHTUNG

Hier muss man allerdings aufpassen. Sollten wir in der Library auch schon eine Activity mit einem Layout bauen, die das eigene Widget mit den eigenen Attributen benutzt, müssen wir den Namensraum in diesem Layout auf das Package der Library setzen, um Übersetzungsfehler zu vermeiden, denn der Namensraum bestimmt, in welcher Ressource nach den Attributen gesucht wird. Benutzen wir nun die Library in einem Projekt (als Android-Library), dann stimmt für das Layout der Namensraum nicht mehr, und es kommt zu Übersetzungsfehlern innerhalb der Anwendung. Um die Übersetzungsfehler auszumerzen, müssen wir das Layout aus der Library in die Anwendung *duplizieren* und den Namensraum entsprechend ändern, dann können wir die Activity aus der Library verwenden. Das ist allerdings nicht unbedingt im Sinne des Erfinders, und ich denke, das könnte in Zukunft auch noch verbessert werden. Unproblematisch ist es, wenn wir nur die eigene View und keine fertige Activity anbieten und die Layouts in der Anwendung erstellen.

Weitere Methoden sind beim Erstellen eigener View möglicherweise wichtig:

onFinishInflate()	Wird aufgerufen, sobald die View und alle Kindelemente aus dem XML-File geladen und erzeugt worden sind.
protected void onSizeChanged (int w, int h, int oldw, int oldh)	Wird aufgerufen, wenn sich die Größe der View geändert hat.
protected void onMeasure (int widthMeasureSpec, int heightMeasureSpec)	Wird aufgerufen, wenn die View »vermessen« werden soll, hier kann eine View z.B. für WRAPPED_CONTENT die Größe ihres Inhalts ermitteln und die Maße richtig setzen.

Tabelle 3.26: **Weitere wichtige Methoden für eigene Views/Widgets**

Für eine vollständige Implementierung eines eigenen Widgets müssen wir noch die onMeasure(...)-Methode betrachten. Hier können wir unser Widget »ausmessen« und auf die Größe bringen, die wir wollen. In den meisten Fällen reicht die Standardimplementierung der Klasse View aus, aber besonders wenn als Layout-Parameter WRAPPED_CONTENT gewählt wurde, müssen wir in dieser Methode das Ausmass unseres Widgets entsprechend berechnen.

Listing 3.48: **onMeasure(...)-Methode**

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    Log.d(Globals.LOG_TAG,"onMeasure(int widthMeasureSpec, int heightMeasu➥
       reSpec)");
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int modeW = MeasureSpec.getMode(widthMeasureSpec);
```

```
    int modeH = MeasureSpec.getMode(heightMeasureSpec);
    int specW = MeasureSpec.getSize(widthMeasureSpec);
    int specH = MeasureSpec.getSize(heightMeasureSpec);
    int w = this.getSuggestedMinimumWidth();
    int h = this.getSuggestedMinimumHeight();
    Log.d(Globals.LOG_TAG,"Width and Heigh: Suggested "+w+" "+h);
    if (modeW == MeasureSpec.EXACTLY)
    {
        w = specW;
        Log.d(Globals.LOG_TAG,"Width: Exactly "+w);
    }
    if (modeH == MeasureSpec.EXACTLY)
    {
        h = specH;
        Log.d(Globals.LOG_TAG,"Height: Exactly "+h);
    }

    if (modeW == MeasureSpec.AT_MOST)
    {
        w = specW;
        Log.d(Globals.LOG_TAG,"Width: At most "+w);
    }
    if (modeH == MeasureSpec.AT_MOST)
    {
        h = specH;
        Log.d(Globals.LOG_TAG,"Height: At most "+h);
    }
    if (modeW == MeasureSpec.UNSPECIFIED)
    {
        Log.d(Globals.LOG_TAG,"Width: Unspecified "+w);
    }
    if (modeH == MeasureSpec.UNSPECIFIED)
    {
        Log.d(Globals.LOG_TAG,"Height: Unspecified "+h);
    }

    this.setMeasuredDimension(w, h);
}
```

Entscheidend in der `onMeasure(…)`-Methode ist der Modus, der durch das umliegende Layout bestimmt wird. Mittels der Aufrufe von

```
int modeW = MeasureSpec.getMode(widthMeasureSpec);
int modeH = MeasureSpec.getMode(heightMeasureSpec);
```

ermitteln wir den jeweiligen Modus für die Breite und die Höhe des Widgets, und mittels

```
int specW = MeasureSpec.getSize(widthMeasureSpec);
int specH = MeasureSpec.getSize(heightMeasureSpec);
```

ermitteln wir die Größe als Hinweis auf die gewünschte Größe. Anhängig vom Modus müssen wir Breite und Höhe ggf. noch weiter behandeln:

MeasureSpec.AT_MOST	Maximal verfügbare Größe. Dieser Modus wird z.B. bei WRAPPED_CONTENT übergeben, wenn das Layout den verfügbaren Platz bereits weiß, und wir können die Größe des Widgets unter Berücksichtigung der verfügbaren Größe berechnen. Das Widget **darf nicht** größer werden, **kann** aber kleiner sein.
MeasureSpec.EXACTLY	Das Widget muss exakt diese Größe haben. Wird z.B. für MATCH_PARENT übergeben, wenn das Layout den verfügbaren Platz für das Widget kennt oder eine feste Größe für das Widget angegeben ist.
MeasureSpec.UNSPECIFIED	Das Layout will wissen, wie groß das Widget werden kann. Das wird vom Layout dann übergeben, wenn z.B. WRAPPED_CONTENT angegeben und der verfügbare Platz für das Widget noch nicht bekannt ist, z.B. wenn mit dem Parameter android:layout_weight gearbeitet wird, um eine anteilige Verteilung zu erreichen. Wir können die Größe des Widgets z.B. anhand des Inhalts bestimmen und entsprechend setzen, das Layout weiß dann, wie viel Platz unser Widget tatsächlich beansprucht.

Tabelle 3.27: **Messmethoden in onMeasure(...)**

Die oben beschriebenen Messmethoden müssen wir jeweils für Höhe und Breite ausführen.

Die Methode kann, bis alle Maße feststehen, durchaus mehrmals aufgerufen werden. Das passiert dann, wenn sich durch die Berechnung der Ausmaße eines Widgets die Bedingungen für die anderen Widgets ändern, und tritt meist auf, wenn innerhalb des Layouts mit dem Attribut android:layout_weight gearbeitet wird.

In diesem Beispiel haben wir eine View abgeleitet und durch das Überschreiben der onDraw(...)-Methode sowie eines Event-Handlers spezialisiert.

Wenn wir zusammengesetzte Widgets erstellen möchten, müssen wir das über eine `ViewGroup` realisieren, in der wir Kindelemente einhängen können. Da mit den Layouts wie LinearLayout, FrameLayout etc. bereits ViewGroups mit entsprechender Funktionalität zur Verfügung stehen, leiten wir zusammengesetzte Widgets in der Regel von einem bestehenden Layout ab.

Android bedient sich dieser Technik z.B. für Spinner-Widgets und das AutoCompleteTextView-Widget.

In der Spielwiese benutzen wir die Technik, um ein Eingabefeld mit vorangestelltem Text (einem Label) zu erstellen. Dabei soll das zusammengesetzte Element selbst aussehen wie ein `EditText`-Widget.

Die Klasse nennen wir `LabeledEditText` und leiten diese von `LinearLayout` ab. In den allermeisten Fällen ist das `LinearLayout` eine gute Basisklasse für zusammengesetzte Widgets.

Benutzt wird unser Widget später folgendermaßen:

Listing 3.49: **Auszug aus dem Layout**

```
[…]
<de.androidpraxis.SpielwieseLibrary3.widgets.LabeledEditText
android:layout_width="match_parent"
android:layout_height="wrap_content"
app:label="@string/labeledittextlabel"
/>
[…]
```

Damit wir das Label unseres Widgets setzen können, definieren wir ein entsprechendes Attribut für unsere Klasse:

Listing 3.50: **Deklaration des Attributs in attrs.xml**

```
[…]
<declare-styleable name="LabeledEditText">
    <attr name="label" format="string"/>
</declare-styleable>
[…]
```

Die Klasse sieht wie folgt aus. Hier schauen wir uns nur den wichtigsten Bestandteil an, in dem das zusammengesetzte Widget erzeugt wird:

```
public class LabeledEditText extends LinearLayout {
private EditText textEdit = null;
private TextView label = null;
public LabeledEditText(Context context) {
    super(context);
    init(null,-1);
}
public LabeledEditText(Context context, AttributeSet attrs) {
    super(context, attrs);
    init(attrs,-1);
}
public LabeledEditText(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
    init(attrs, defStyle);
}

private void init(AttributeSet attrs, int defStyle)
{
    this.setOrientation(HORIZONTAL);
    this.setAddStatesFromChildren(true);
```

Mit diesem Abschnitt initialisieren wir das Layout mit horizontaler Ausrichtung und der Replikation der Zustände der eingehängten Widgets. Als Nächstes erstellen wir die Widgets für das Label und die Texteingabe:

```
    TypedArray a = getContext().obtainStyledAttributes(attrs,R.styleable.
LabeledEditText,defStyle,0);
    CharSequence labelText = a.getText(R.styleable.LabeledEditText_label );
    label = new TextView(getContext());
    if (labelText!=null) setLabel(labelText);
    textEdit = new EditText(getContext());
    addView(label,new LinearLayout.LayoutParams(LayoutParams.WRAP_CONTENT,
LayoutParams.MATCH_PARENT));
    addView(textEdit,new LinearLayout.LayoutParams(0,LayoutParams.MATCH_PA➥
        RENT ,1));
    label.setBackgroundResource(0);
    textEdit.setBackgroundResource(0);
```

Durch die `LinearLayout.LayoutParams`-Klasse setzen wir die Größe unserer Widgets innerhalb des Layouts. Über `setBackground(0)` sorgen wir dafür, dass die inneren Widgets keinen Hintergrund haben.

```
    this.setBackgroundResource(android.R.drawable.edit_text);
```

Diese Zeile ist die Magie in Verbindung mit `setAddStatesFromChildren(true)`. Die Ressource `android.R.drawable.edit_text` referenziert die `StateListDrawable`-Ressource von Android, die für die Eingabefelder benutzt wird. Wir setzen diesen Hintergrund, nachdem wir die Hintergründe der eingehängten Widgets ausgeschaltet haben, für unser neues, zusammengesetztes Widget. Unser zusammengesetztes Widget erhält seinen Zustand aus den Zuständen der eingehängten Widgets, und damit sieht es nachher aus wie ein Eingabefeld, allerdings um unser Label erweitert.

Listing 3.51: **Zusammengesetztes Widget mit Layout-Tricks**

```
}
[…]
}
```

TIPP

Was es mit den `StateListDrawable`-Ressourcen auf sich hat, erfahren wir später, wenn wir uns mit den Styles und Themes beschäftigen.



Abbildung 3.25: **Das Ergebnis der Klasse LabeledEditText**

Die hier abgebildete TextView, und mit ihr auch die TextView, die wir als Label (hier: *Das Label* und *Ein weiteres Feld*) in dem zusammengesetzten Widget benutzen, sowie die Buttons sind über Styles und das gewählte Theme in der Spielwiese definiert:

Listing 3.52: **Stil der TextView und der Buttons**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="MeineTextView">
        <item name="android:textAppearance">@style/MeineTextViewTextAppea➥
            rance</item>
        <item name="android:gravity">center</item>
        <item name="android:background">@color/meinetextview_background</
item>
    </style>
    <style name="MeineTextViewTextAppearance">
        <item name="android:textSize">28sp</item>
        <item name="android:textColor">@color/meinetextview_color</item>
    </style>

    <style name="MeinButtonStyle" parent="android:Widget.Button">
        <item name="android:background">@drawable/mybutton</item>
        <item name="android:textColor">@color/mybutton_color</item>
        <item name="android:textSize">28sp</item>
    </style>
</resources>
[…]
<?xml version="1.0" encoding="utf-8"?>
```

```
<resources>
    <style parent="android:Theme.Holo.Light" name="MeinTheme">
        <item name="android:textViewStyle">@style/MeineTextView</item>
        <item name="android:buttonStyle">@style/MeinButtonStyle</item>
    </style>
</resources>
```

3.11.8 Dialoge und Benachrichtigungen

Ein essenzielles Element aller Benutzeroberflächen sind Dialoge, die eine aktuelle Aktivität »unterbrechen« und etwas am Bildschirm anzeigen oder zu einer Eingabe auffordern. Das Wesen des Dialogs ist, dass er erst auf Bestätigung geschlossen wird oder, im Falle eines Fortschrittsanzeigedialogs, die Aktion fertiggestellt oder abgebrochen wurde.

Android bringt eine Basisklasse für Dialoge mit und auf dieser aufbauend Standarddialoge für folgende Einsatzzwecke:

Dialog	Basisklasse für alle möglichen Formen der Dialoge, Basis für eigene Dialoge.
AlertDialog	Ein Standarddialog für Meldungen und eine Liste mit auswählbaren Einträgen sowie keine bis drei Buttons. Kann z.B. für Sicherheitsabfragen à la »Wollen Sie den Eintrag wirklich löschen?« oder aber auch zur Auswahl von Optionen aus einer Liste von Möglichkeiten genutzt werden. Der AlertDialog wird als Basis für die meisten Dialoge dieser Art empfohlen.
ProgressDialog	Der ProgressDialog dient dazu den Fortschritt einer laufenden Operation als Prozentbalken oder drehendes Rads darzustellen. Könnte z.B. für »Anmeldung läuft ...« eingesetzt werden oder auch beim Download/Upload von Daten, sofern man das nicht sowieso im Hintergrund erledigen möchte.
DatePickerDialog und TimePickerDialog	Dialoge um ein Datum und/oder eine Uhrzeit auszuwählen
AlertDialog.Builder	Hilfsklasse, um einen AlertDialog zusammenzubauen
DialogFragment	Ein Fragment, das sich wie ein Dialog verhält.

Tabelle 3.28: **Dialog-Klassen**

Dialoge werden in der Regel im Kontext einer Activity ausgeführt. Die Activity implementiert folgende Methoden, damit wir Dialoge erstellen und nutzen können:

protected Dialog onCreateDialog(int id) protected Dialog onCreateDialog (int id, Bundle args)	Wird aufgerufen, wenn der Dialog mit der übergebenen ID erstellt werden soll. Alternativ werden Argumente in einem Bundle übergeben.
protected void onPrepareDialog (int id, Dialog dialog) protected void onPrepareDialog (int id, Dialog dialog, Bundle args	Wird aufgerufen, bevor der Dialog angezeigt wird. Hier können kontextabhängige Aktionen stattfinden, um bestimmte Optionen zu sperren o.Ä.
public final void showDialog(int id) public final void showDialog(int id, Bundle args)	Aufrufen des Dialogs
public final void dismissDialog(int id)	Schließen des Dialogs
DialogInterface.OnClickListener	Reagieren auf das Klicken von Buttons oder Listeneinträge

Tabelle 3.29: **Dialog-Methoden der Activity**

An den Methoden ist zu erkennen, dass wir Dialoge innerhalb der Activity über eine ID ansprechen. Die IDs deklarieren wir als Konstanten innerhalb der Activity oder ggf. als globale Konstanten.

Listing 3.53: **Deklaration der Dialog ID**

```
public class EditMediaActivity extends Activity implements DialogInterface.
OnDismissListener, DialogInterface.OnCancelListener {
    private static final int EDIT_DIALOG_ID = 1;
    […]
}
```

Die EditMediaActivity benutzt einen kleinen Trick, um eine Activity komplett zu einem Dialog umzufunktionieren, indem der Dialog direkt beim Erzeugen der Activity angezeigt wird:

Listing 3.54: **Anzeigen des Dialogs**

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    showDialog(EDIT_DIALOG_ID);
}
```

Normalerweise legt sich eine Activity allerdings mit einem durch das gewählte Theme festgelegten Hintergrund über den Bildschirm, sodass der Dialog nicht wie ein Dialog aussehen würde. Um die Activity transparent zu gestalten müssen wir im Manifest das Theme `Theme.Translucent.NoTitleBar` auswählen:

```
<activity android:name="de.androidpraxis.SpielwieseLibrary3.EditMediaActivi➥
    ty" android:theme="@android:style/Theme.Translucent.NoTitleBar" >
```

ACHTUNG

Leider klappt das explizite Setzen des Themes mittels `setTheme(android.R.style.Theme_Translucent_NoTitleBar)` **nicht. Der Hintergrund der Activity ist dann immer noch nicht transparent.**

Damit nun aber unser Dialog überhaupt angezeigt werden kann, muss er in onCreateDialog(...) auch erstellt werden:

Listing 3.55: **Erstellen des Dialogs**

```
@Override
protected Dialog onCreateDialog(int id) {
    Dialog dialog;
    switch(id) {
    case EDIT_DIALOG_ID:
        Context context = this;
        dialog = new EditDialog(context);
    break;

    default:
        dialog = null;
    }
    if (dialog!=null)
    {
        dialog.setOnCancelListener(this);
        dialog.setOnDismissListener(this);
    }
    return dialog;
}
```

Wenn wir mehr Dialog hätten, würden wir das über weitere IDs erledigen.

In diesem Beispiel erstellen wir einen Dialog vom Typ `EditDialog`, den wir selbst definieren und der direkt von `Dialog` abgeleitet ist:

Listing 3.56: **Die Deklaration des eigenen Dialogs**

```
private class EditDialog extends Dialog implements OnClickListener
{
private View dialogView;

public EditDialog(Context context, boolean cancelable,
OnCancelListener cancelListener) {
[…]
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    dialogView = getLayoutInflater().inflate(R.layout.editmediadialog,
null);
    setContentView(dialogView);
    setTitle("Bearbeiten");

    Button btn = (Button)dialogView.findViewById(R.id.button_ok);
    btn.setOnClickListener(this);

    btn = (Button)dialogView.findViewById(R.id.button_cancel);
```

```
        btn.setOnClickListener(this);

        mapDataToView();
    }

    public void mapDataToView()
    {
    […]
    }

    public void mapDataFromView()
    {
    […]
    }

    @Override
    public void onClick(View view) {

        switch (view.getId())
        {
            case R.id.button_ok:
            mapDataFromView();
            dismiss();
        break;

        case R.id.button_cancel:
            cancel();
            break;
        }
    }
    };
```

Mittels

```
dialogView = getLayoutInflater().inflate(R.layout.editmediadialog, null);
setContentView(dialogView);
```

wird das Layout des Dialogs aus der Layout-Ressource geladen. Im Weiteren werden die Buttons mit dem Dialog als onClick()-Listener verknüpft, um auf das Drücken von *Speichern* oder *Abbrechen* zu reagieren.

Die Methoden mapDataToView() und mapDataFormView() sind hier gekürzt, diese haben wir eingeführt, um beim Erstellen des Dialogs die übergebenen Daten in die Widgets zu übertragen und beim Betätigen von *Speichern* die Eingaben wiederum in die Datenbank zurückzuschreiben.

Durch die Methoden Dialog.dismiss() und Dialog.cancel() wird der Dialog nun geschlossen. Normalerweise geht die Kontrolle dann wieder an die Activity zurück, die wir jedoch nur als »Träger« des Dialogs konzipiert haben. Wird der Dialog geschlossen, dann wollen wir auch die Activity beenden, damit wir nicht einfach auf einem transparenten Etwas sitzen bleiben.

Das erreichen wir, in dem wir in der Activity die Schnittstellen OnDismissListener und On-CancelListener implementieren und die Activity beim Dialog entsprechend anmelden:

```
dialog.setOnCancelListener(this);
dialog.setOnDismissListener(this);
```

Die Reaktion fällt dann relativ kurz aus:

Listing 3.57: **Reaktion auf Cancel bzw. das Schließen des Dialogs**

```
@Override
public void onCancel(DialogInterface dialog) {
    finish();
}

@Override
public void onDismiss(DialogInterface dialog) {
    finish();
}
```

Da die Activity als Listener angemeldet ist können wir auf das Schließen des Dialogs reagieren und per `finish()` auch unsere Activity beenden.

Das vorliegende Beispiel können wir aber genauso gut benutzen, wenn die Activity selbst Funktionalität besitzt, dann würden wir auf das `finish()` verzichten.

Für die übrigen, speziellen Dialoge bleibt der Rahmen der Activity im Grunde aber immer gleich. Wir definieren IDs für die unterschiedlichen Dialoge und benutzen showDialog(...) zum Anzeigen derselben. Was sich unterscheidet, ist die Erstellung der Dialoge, der Alert-Dialog liefert z.B. einen Builder mit, den wir für das Zusammenbauen eines Alert-Dialogs benutzen können.

Listing 3.58: **Benutzen eines Alert-Dialogs für einfache Dialoge**

```
protected class ConfirmDeleteDialog extends DialogFragment implements
DialogInterface.OnClickListener
{
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {

    return new AlertDialog.Builder(getActivity())
    .setIcon(R.drawable.icon)
    .setTitle(R.string.shouldImageBeDeleted)
    .setMessage(R.string.shouldImageBeDeletedText)
    .setPositiveButton(R.string.yes,this)
    .setNegativeButton(R.string.no,this)
    .create();
}

@Override
public void onClick(DialogInterface dialog, int which) {
switch (which)
{
    case DialogInterface.BUTTON_POSITIVE:
```

```
        ShowImageFragment.this.deleteImage();
    break;
    case DialogInterface.BUTTON_NEGATIVE:
    break;
    case DialogInterface.BUTTON_NEUTRAL:
    break;
    default:
    //Hier könnte ein Item ausgewählt worden sein!
}
}
};
```

Das obige Beispiel arbeitet in einem Fragment und ist daher als DialogFragment realisiert. Das Zusammenbauen des eigentlichen Dialogs erledigen wir hier aber genauso, wie wir das in einer Activity in der Methode `onCreateDialog(…)` tun würden.

Der Builder ist so aufgebaut, dass wir durch das Aneinanderhängen der Methodenaufrufe den Dialog bequem mit einem Icon, einem Titel, einer Nachricht sowie zwei Buttons bestücken können. Zu den Buttons können wir dann direkt noch einen OnClick-Listener mitgeben, in dem wir auf das Betätigen der Buttons reagieren können.

INFO

In den SDK-Beispielen finden wir sehr häufig anonyme Listener (`new OnClickListener() { … }`), die dort übergeben werden, ich persönlich mag das an dieser Stelle nicht so sehr und bevorzuge die oben gewählte Variante, das DialogFragment bzw. den Dialog selbst als Listener zu implementieren. Das ist auch vollkommen problemlos möglich, da die Buttons eine negative ID haben (BUTTON_POSITIVE = -1...) und somit keine Konflikte beim Anklicken von Listeneinträgen entstehen.

Das DialogFragment können wir innerhalb unseres Fragments dann wie folgt benutzen:

Listing 3.59: **Benutzen des Dialogs**

```
protected void deleteImage() {
    Toast.makeText(getActivity(), "Löschen ist nicht implementiert", Toast.
LENGTH_SHORT).show();
}
protectedvoid askForDeleteImage()
{
    ConfirmDeleteDialog dlg = new ConfirmDeleteDialog();
    dlg.show(getFragmentManager(),"confirmDeleteDialog");
}
```

Die Methode `deleteImage()`, die hier absolut sicher ist ;-), wird vom OnClick-Listener unseres Dialogs aufgerufen. Der Dialog selbst wird innerhalb von `askForDeleteImage()` erzeugt und dargestellt.

Der AlertDialog kann aber noch viel mehr. Über Arrays, Cursor oder Adapter können wir statt eines Textes eine Auswahl aus Einträgen bereitstellen. Als Beispiel habe ich das Verschieben von Bildern in einen anderen Ordner benutzt. Als Quelle für die Ordnernamen dienen die Ordner, die der Medienscanner bereits erkannt hat. Hierin enthalten ist auch ein

kleiner, feiner Trick, mit dem man die einzelnen Ordnernamen aus dem MediaStore herausfinden bzw. generell ein SQL-Statement mit einem »Group By« in die Cursor-Adapter einflicken kann. Aus unerfindlichen Gründen werden die Ordnernamen nämlich nicht normalisiert, d.h., es gibt im MediaStore keine Tabelle (oder ich habe sie noch nicht gefunden), die nur die Albumnamen enthält. Diese sind einfach als Textfeld im MediaStore gespeichert.

Listing 3.60: **Einen Adapter an den AlertDialog binden**

```
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {

return new AlertDialog.Builder(getActivity())
    .setIcon(R.drawable.icon)
    .setTitle(R.string.moveImageToFolder)
    .setSingleChoiceItems(cursorAdapterAlbums, 0, this)
    .setPositiveButton(R.string.exec,this)
    .setNegativeButton(R.string.cancel,this)
    .create();
}
@Override
public void onClick(DialogInterface dialog, int which) {
    switch (which)
    {
    case DialogInterface.BUTTON_POSITIVE:
        if (selectedItem != -1)
        {
          Cursor c = (Cursor)cursorAdapterAlbums.getItem(selectedItem);
          ShowImageFragment.this.moveImage(c.getString(1));
        }
    break;
    case DialogInterface.BUTTON_NEGATIVE:
    break;
    case DialogInterface.BUTTON_NEUTRAL:
    break;
    default:
    //Hier könnte ein Item ausgewählt worden sein!
        selectedItem = which;
    /*Cursor c = (Cursor)cursorAdapterAlbums.getItem(which);
    ShowImageFragment.this.moveImage(c.getString(1));
    dialog.dismiss();*/
    }
}
```

Hier binden wir den Adapter an den Dialog, und zwar mittels singleChoiceItems(...). Dadurch wird neben jedem Eintrag ein Radio-Button angezeigt, mit dem man den Eintrag anhaken kann. Erst bei Betätigen des Buttons *Ausführen* wird dann die Aktion wirklich ausgeführt. Wir führen das Verschieben nicht sofort beim Auswählen des Eintrags aus, um Fehleingaben zu vermeiden.

Damit in der ListView der Radio-Button angezeigt wird, muss diese entsprechend konfiguriert werden:

```
cursorAdapterAlbums = new SimpleCursorAdapter(getActivity(),
android.R.layout.simple_list_item_single_choice,
    null,
    new String[] { MediaStore.Images.Media.BUCKET_DISPLAY_NAME } ,
    new int[] {android.R.id.text1});
getLoaderManager().initLoader(0, null, this);
```

Das Entscheidende ist hier dem Adapter die Ressource `android.R.layout.simple_list_item_single_choice` für den Listeneintrag mitzugeben.

Ach so, und hier ist noch der Trick um die Albumnamen herauszufiltern:

Listing 3.61: **Laden der Daten für den Alert-Dialog**

```
@Override
public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1) {
return new
    CursorLoader(getActivity(),MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
new String[] { MediaStore.Images.Media._ID, MediaStore.Images.Media.BU➥
    CKET_DISPLAY_NAME }, "1=1) group by ("+MediaStore.Images.Media.BUCKET_
    DISPLAY_NAME, null, null);
}
```

AlertDialog	create()	Erzeugt den Dialog für die spätere Verwendung, z.B. in onCreateDialog(int id)
AlertDialog.Builder	setAdapter(ListAdapter adapter, DialogInterface.OnClickListener listener)	Setzt einen Adapter als Datenquelle für die Listeneinträge. **Achtung:** Wenn eine Liste dargestellt werden soll, darf kein Text gesetzt werden.
AlertDialog.Builder	setCancelable(boolean cancelable)	Legt fest, ob der Dialog abgebrochen werden kann.
AlertDialog.Builder	setCursor(Cursor cursor, DialogInterface.OnClickListener listener, String labelColumn)	Setzt einen Cursor als Datenquelle für die Listeneinträge.
AlertDialog.Builder	setCustomTitle(View customTitleView)	Setzt eine eigene View als Überschriftenelement des Dialogs.
AlertDialog.Builder AlertDialog.Builder AlertDialog.Builder	setIcon(Drawable icon) setIcon(int iconId) setIconAttribute(int attrId)	Setzt das Icon.
AlertDialog.Builder	setInverseBackgroundForced(boolean useInverseBackground)	Legt fest, ob der Hintergrund invertiert werden soll.

Tabelle 3.30: **Methoden des Dialog-Builders**

AlertDialog.Builder AlertDialog.Builder	setItems(int itemsId, DialogInterface.OnClick-Listener listener) setItems(CharSequence[] items, DialogInterface.OnClickListener listener)	Setzt die Einträge entweder auf eine String-Array-Ressource oder als Array von CharSequences.
AlertDialog.Builder AlertDialog.Builder	setMessage(CharSequence message) setMessage(int messageId)	Setzt den Text, der im Dialog angezeigt werden soll. Darf nicht aufgerufen werden wenn eine Liste dargestellt werden soll.
AlertDialog.Builder AlertDialog.Builder AlertDialog.Builder	setMultiChoiceItems(CharSequence[] items, boolean[] checkedItems, DialogInterface.OnMultiChoiceClickListener listener) setMultiChoiceItems (Cursor cursor, String isCheckedColumn, String labelColumn, DialogInterface.OnMultiChoiceClick-Listener listener) setMultiChoiceItems(int itemsId, boolean[] checkedItems, DialogInterface.OnMultiChoiceClick-Listener listener)	Setzt die Einträge als Multi-Choice-Items, das heißt es können mehrere Einträge ausgewählt werden.
AlertDialog.Builder AlertDialog.Builder AlertDialog.Builder AlertDialog.Builder AlertDialog.Builder	setPositiveButton(int textId, DialogInterface.OnClickListener listener) setNegativeButton (CharSequence text, DialogInterface.OnClick-Listener listener) setNegativeButton(int textId, DialogInterface.OnClickListener listener) setNeutralButton(int textId, DialogInterface.OnClickListener listener) setNeutralButton (CharSequence text, DialogInterface.OnClick-Listener listener)	Setzen der Buttons.

Tabelle 3.30: **Methoden des Dialog-Builders (Forts.)**

AlertDialog.Builder AlertDialog.Builder AlertDialog.Builder	setOnCancelListener(DialogInterface.OnCancelListener onCancelListener) setOnItemSelectedListener(AdapterView.OnItemSelectedListener listener) setOnKeyListener(DialogInterface.OnKeyListener onKeyListener)	Setzen von Listenern.
AlertDialog.Builder AlertDialog.Builder AlertDialog.Builder AlertDialog.Builder	setSingleChoiceItems(CharSequence[] items, int checkedItem, DialogInterface.OnClickListener listener) setSingleChoiceItems(ListAdapter adapter, int checkedItem, DialogInterface.OnClickListener listener) setSingleChoiceItems(int itemsId, int checkedItem, DialogInterface.OnClickListener listener) setSingleChoiceItems(Cursor cursor, int checkedItem, String labelColumn, DialogInterface.OnClickListener listener)	Setzen von Einträgen als SingleChoiceItems, das heißt neben den Einträgen wird ein Radio-Button dargestellt, und es kann ein Eintrag gewählt werden.
AlertDialog.Builder AlertDialog.Builder	setTitle(CharSequence title) setTitle(int titleId)	Setzt die Überschrift des Dialogs.
AlertDialog.Builder	setView(View view)	Setzt ein eigenes Layout als Dialoginhalt. Damit ist es möglich, den Builder auch für komplett eigene Dialoge zu nutzen.
AlertDialog	show()	Erstellen und Anzeigen des Dialogs.

Tabelle 3.30: **Methoden des Dialog-Builders (Forts.)**

Der ProgressDialog bietet entweder einen Fortschrittsbalken oder ein drehendes Rad als Indikator für einen lang laufenden Prozess, dessen Umfang und Fortschritt man nicht kennen kann.

Listing 3.62: **Einsatz eines ProgressDialog in einem AsyncTask**

```
private class SimulateLongRunningTask extends AsyncTask<Integer, Integer,
Long> {
    private ProgressDialog progressDialog;
```

```
    protected void onPreExecute()
    {
        progressDialog = new ProgressDialog(TaskAndServicesActivity.this);
        progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
        progressDialog.setMessage("Machwas...");
        progressDialog.setCancelable(false);
        progressDialog.show();
    }
    protected Long doInBackground(Integer... steps) {
        int count = steps[0];
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
        publishProgress((int) ((i / (float) count) * 100));
        try {
        Thread.sleep(100);
        } catch (InterruptedException e) {
        break;
        }
        }
        return totalSize;
    }
    protected void onProgressUpdate(Integer... progress) {
        progressDialog.setProgress(progress[0]);
    }
    protected void onPostExecute(Long result) {
        progressDialog.dismiss();
        TaskAndServicesActivity.this.finish();
    }
}
```

In diesem Beispiel simulieren wir eine asynchrone länger laufende Aufgabe, die den Fortschritt in einem ProgressDialog darstellt.

Um einen einfachen "Warten..."-Dialog zu erstellen, reicht dagegen das Konstrukt

```
ProgressDialog dialog = ProgressDialog.show(MyActivity.this, "", "Bitte war➥
    ten…", true);
dialog.show();
[…]
dialog.dismiss();
```

aus.

Die Dialoge zur Datums- und Uhrzeitauswahl sind ähnlich einfach zu nutzen. Innerhalb von obCreateDialog(...) erstellen wir einfach eine Instanz des Dialogs und setzen den entsprechenden Listener um auf die Auswahl des Datums oder der Zeit zu reagieren.

DatePickerDialog(Context context, DatePickerDialog.OnDateSetListener callBack, int year, int monthOfYear, int dayOfMonth) DatePickerDialog(Context context, int theme, DatePickerDialog.OnDateSetListener callBack, int year, int monthOfYear, int dayOfMonth)		Erstellen des Dialogs mit Setzen des Listeners und des aktuell anzuzeigenden Datums.
void	updateDate(int year, int monthOfYear, int dayOfMonth)	Setzen des im Dialog anzuzeigenden Datums.
abstract void	onDateSet(DatePicker view, int year, int monthOfYear, int dayOfMonth)	Methode des Listeners, die beim Setzen des Datums aufgerufen wird.

Tabelle 3.31: **Wichtige Methoden der Klasse DatePickerDialog**

TimePickerDialog(Context context, TimePickerDialog.OnTimeSetListener callBack, int hourOfDay, int minute, boolean is24HourView) TimePickerDialog(Context context, int theme, TimePickerDialog.OnTimeSetListener callBack, int hourOfDay, int minute, boolean is24HourView)		Erstellen des Dialogs mit dem Listener und der anzuzeigenden Uhrzeit.
void	updateTime(int hourOfDay, int minutOfHour)	Setzen der anzuzeigenden Uhrzeit.
abstract void onTimeSet(TimePicker view, int hourOfDay, int minute)		Methode des Listeners, die beim Setzen der Uhrzeit aufgerufen wird.

Tabelle 3.32: **Wichtige Methoden der Klasse TimePickerDialog**

Neben den Dialogen, die in der Regel eine Unterbrechung der aktuellen Aktivität bedeuten bzw. eine direkte Benutzerreaktion erfordern gibt es noch Benachrichtigungsmechanismen, die über Ereignisse informieren und keine direkte Benutzerinteraktion erforderlich machen oder erforderlich machen sollen.

Wir haben dafür zwei Möglichkeiten zur Verfügung:

1. Toast Notifications
2. Status Bar Notifications

Die Toast-Notification ist eine Benachrichtigung, die innerhalb einer Activity als Reaktion auf eine Benutzerinteraktion benutzt wird, z.B. wenn etwas gespeichert oder gelöscht wurde. In diesem Fall wollen wir den Abschluss dieser Aktion, die nur einen kurzen Zeitraum dauert, als Nachricht präsentieren:

```
Toast.makeText(this, "Die Notiz wurde gespeichert", Toast.LENGTH_SHORT).
show();
```

Mehr braucht man nicht zu tun, durch diesen Aufruf wird die Meldung kurz an der Standardposition für Toasts (in der Regel mittig im unteren Drittel des Bildschirms) eingeblendet.

Wir können die Position aber auch beeinflussen:

```
Toast toast = Toast.makeText(this, "Die Notiz wurde gespeichert", Toast.
LENGTH_SHORT);
toast.setGravity( Gravity.TOP|Gravity.LEFT,0,0);
toast.show();
```

Mittels `setGravity(<Gravity>,<Offset X>,<Offset Y>)` können wir den Toast quasi frei auf dem Schirm platzieren.

Wenn uns der Standard-Toast zu langweilig ist, können wir den Toast auch mit einem eigenen Layout füttern. In diesem Fall erstellen wir den Toast nicht mittels makeText(...), sondern mittels des Konstruktors und setzen ein eigenes Layout als View ein:

Listing 3.63: **Layout für einen eigenen Toast**

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/toast_layout_root"
android:orientation="horizontal"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:padding="10dp"
android:background="#DAAA"
>
    <ImageView android:id="@+id/image"
     android:layout_width="wrap_content"
     android:layout_height="fill_parent"
     android:layout_marginRight="10dp"
     />
    <TextView android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:textColor="#FFF"
    />
</LinearLayout>
```

Das eigene Layout holen wir dann mit dem `LayoutInflater` und setzen es innerhalb des Toasts. Am besten packen wir das in eine eigene Toast-Klasse, um das Ganze flexibel und wiederverwendbar zu gestalten:

Listing 3.64: **Einen eigenen Toast erstellen und benutzen**

```
public class CustomToast extends Toast {
    public CustomToast(Context context) {
        super(context);
    }
    public static CustomToast makeCustomToast(Context context, int imageRes,
CharSequence text, int duration )
    {
        LayoutInflater inflater = (LayoutInflater)context.
getSystemService(Context.LAYOUT_INFLATER_SERVICE);
```

```
        View layout = inflater.inflate(R.layout.toast_layout,null);

        ImageView imageView = (ImageView) layout.findViewById(R.id.image);
        imageView.setImageResource(imageRes);

        TextView textView = (TextView) layout.findViewById(R.id.text);
        textView.setText(text);

        CustomToast toast = new CustomToast(context);
        toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);
        toast.setDuration(duration);
        toast.setView(layout);
        return toast;
}

}
[…]
Toast toast = CustomToast.makeCustomToast(getApplicationContext(),R.draw➥
    able.icon,"Die Verarbeitung wurde abgeschlossen!",Toast.LENGTH_LONG);
toast.show();
```

Die Toasts dienen wie beschrieben dem direkten Anzeigen eines Ereignisses, in der Regel nach einer Benutzerinteraktion, um etwas ganz klarzumachen, aber ohne dass der Benutzer diese Nachricht noch mal bestätigen muss.

Es gibt aber auch Ereignisse, die zu einem beliebigen Zeitpunkt auftreten können, z.B. Ereignisse, die von Hintergrundaufgaben ausgelöst werden. Dazu zählen z.B. die Benachrichtigungen über eingegangene Nachrichten, verpasste Anrufe, anstehende Termine, fertiggestellte Downloads o.Ä. Diese Ereignisse treten also nicht direkt nach einer Benutzerinteraktion auf, sondern erst wenn ein bestimmtes Datum erreicht, der Nachrichteneingang geprüft wurde oder ein Hintergrundprozess etwas gestartet oder fertiggestellt hat. Darüber hinaus können diese Ereignisse auch auftreten, wenn der Benutzer gar nicht am Gerät präsent ist.

Das bedeutet, dass diese Ereignisse nicht mittels Dialog oder Toast signalisiert werden dürfen, denn entweder macht der Benutzer gerade etwas anderes, z.B. telefonieren, und will dabei nicht unterbrochen werden, oder er ist nicht da und würde einen Toast gar nicht bemerken.

Für diese Situation gibt es die Status Bar Notification, mit der wir in der Status Bar eine Benachrichtigung anheften und, wenn der Benutzer auf die Nachricht klickt, zu einer entsprechenden Activity wechseln können.

Um Benachrichtigungen in der Status Bar zu verwalten, benötigen wir eine Instanz des NotificationManager. Der NotificationManager ist ein Systemservice, den wir mit dem Aufruf von

```
NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
```

anfordern und nutzen können. Eine Benachrichtigung selbst wird durch ein Notification-Objekt repräsentiert.

Ähnlich wie der AlertDialog bringt die Notification einen Builder mit, der uns beim Zusammenbauen einer Notification zur Hand geht:

Listing 3.65: **Den Notification.Builder benutzen**

```
Notification notification = new Notification.Builder(context)
.setSmallIcon(icon)
.setTicker(tickerText)
.setContentTitle(contentTitle)
.setContentText(contentText)
.setWhen(when)
.getNotification();
```

Jede Benachrichtigung wird mit einer ID an den NotificationManager übergeben, so dass wir unsere Nachrichten auch explizit wieder löschen oder die Benachrichtigung im Laufe eines Prozesses austauschen können:

```
notificationManager.notify(NOTIFICATION_ID, notification);
```

Um eine Benachrichtigung zu löschen, benutzen wir den Aufruf:

```
notificationManager.cancel(NOTIFICATION_ID);
```

Innerhalb eines Service kann das wie folgt aussehen:

Listing 3.66: **Setzen einer Benachrichtigung mit Intent**

```
private void createStausBarNotification(CharSequence text)
{
    NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);

    int icon = R.drawable.icon;
    CharSequence tickerText = text;
    long when = System.currentTimeMillis();
    Context context = this;

    CharSequence contentTitle = "Spielwiese Hintergrundservice";
    CharSequence contentText = text;

    Intent notificationIntent = new Intent(this, StartedFromNotificationAc➥
       tivity.class);
    PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notifi➥
       cationIntent, 0);

    Notification notification = new Notification.Builder(context)
    .setSmallIcon(icon)
    .setTicker(tickerText)
    .setContentTitle(contentTitle)
    .setContentText(contentText)
    .setWhen(when)
    .setContentIntent(contentIntent)
    .getNotification();
    notificationManager.notify(NOTIFICATION_ID, notification);
}
```

Bei dieser Benachrichtigung setzen wir ein Intent zum Aufrufen einer Activity. Das erledigen wir mit einem PendingIntent. PendingIntents können überall dort benutzt werden, wo Intent-Objekte zur späteren Verwendung aufgehoben werden müssen:

```
Intent notificationIntent = new Intent(this, StartedFromNotificationActivi➥
    ty.class);
PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notifica➥
    tionIntent, 0);
```

Die StatusBar-Notifications sind nicht sehr kompliziert umzusetzen, bieten aber eine gute Benutzerführung bei länger laufenden Prozessen, die im Hintergrund ablaufen, und eine elegante Möglichkeit, Ereignisse zu melden und aufzuheben, bis der Benutzer Zeit hat, darauf zu reagieren.

Notification getNotification()	Liefert die erzeugte Notification zurück.
Notification.Builder setAutoCancel(boolean autoCancel)	Legt fest, ob die Notification automatisch entfernt wird, wenn der Benutzer auf das Benachrichtigungspanel klickt. Wenn ein OnDelete-Intent gesetzt wurde, wird dieses Intent zu diesem Zeitpunkt ausgeführt.
Notification.Builder setContent(RemoteViews views)	Definiert ein eigenes Layout für die Notification.
Notification.Builder setContentInfo(CharSequence info)	Legt den Text rechts von der Benachrichtigung fest.
Notification.Builder setContentIntent(PendingIntent intent)	Legt das Intent fest, das beim Klicken auf die Benachrichtigung ausgeführt werden soll.
Notification.Builder setContentText(CharSequence text)	Setzt den Text der Benachrichtigung.
Notification.Builder setContentTitle(CharSequence title)	Setzt die Überschrift der Benachrichtigung.
Notification.Builder setDefaults(int defaults)	Setzt Standardwerte: DEFAULT_SOUND DEFAULT_VIBRATE DEFAULT_LIGHTS oder DEFAULT_ALL
Notification.Builder setDeleteIntent(PendingIntent intent)	Setzt das Intent, das aufgerufen wird, wenn der Benutzer die Benachrichtigung vom Panel löscht (z.B. durch das Anklicken des Schließen-Buttons oder wenn der Benutzer alle Benachrichtigungen löscht).

Tabelle 3.33: **Methoden des Notification.Builder**

Notification.Builder setFullScreen-Intent(PendingIntent intent, boolean highPriority)	Setzt ein Intent, das ausgeführt wird, sobald die Benachrichtigung zugefügt wird. Das sollte nur für wirklich wichtige Dinge geschehen und vom Benutzer auch abzustellen sein, denn diese Form der Benachrichtigung unterbricht den Benutzer ja bei seiner momentanen Beschäftigung.
Notification.Builder setLargeIcon(Bitmap icon)	Setzt das große Icon, wenn die Benachrichtigung durch Anklicken in einem eigenen Panel angezeigt wird.
Notification.Builder setLights(int argb, int onMs, int offMs)	Setzt die Farbe des Lichts und die Blinkfrequenz.
Notification.Builder setNumber(int number)	Setzt die Zahl, die neben der Benachrichtigung angezeigt werden soll.
Notification.Builder setOngoing(boolean ongoing)	Weist die Benachrichtigung als »laufend« aus, dadurch wird diese nach oben sortiert und kann manuell nicht geschlossen oder entfernt werden.
Notification.Builder setOnlyAlertOnce (boolean onlyAlertOnce)	Bestimmt ob die Benachrichtigungseffekte nur beim ersten Mal abgespielt werden sollen.
Notification.Builder setSmallIcon(int icon, int level)	Setzt das Icon für die Anzeige in der StatusBar.
Notification.Builder setSmallIcon(int icon)	
Notification.Builder setSound(Uri sound)	Setzt den Sound, der abgespielt werden soll.
Notification.Builder setSound(Uri sound, int streamType)	
Notification.Builder setTicker(CharSequence tickerText, RemoteViews views)	Setzt den Ticker-Text. Das ist der Text, der beim ersten Eintrag der Benachrichtigung in der StatusBar durchläuft.
Notification.Builder setTicker(CharSequence tickerText)	
Notification.Builder setVibrate(long[] pattern)	Setzt das Vibrationsmuster.
Notification.Builder setWhen(long when)	Setzt den Zeitstempel zur Benachrichtigung. Benachrichtigungen werden nach dem Zeitstempel sortiert.

Tabelle 3.33: **Methoden des Notification.Builder (Forts.)**

Wenn wir mit diesen Methoden ein Notification-Objekt erzeugt haben, kommen folgende Methoden des NotificationManagers zum Einsatz:

void cancel(int id) void cancel(String tag, int id)	Entfernen einer vorher zugefügten Benachrichtigung. Wenn ein Tag benutzt wird, dann wird die einzelne Benachrichtigung durch die Kombination aus Tag und ID eindeutig.
void cancelAll()	Entfernen aller Benachrichtigungen.
void notify(int id, Notification notification) void notify(String tag, int id, Notification notification)	Setzen einer Benachrichtigung. Wenn ein Tag benutzt wird, dann wird die einzelne Benachrichtigung durch die Kombination aus Tag und ID eindeutig.

Tabelle 3.34: **Methoden des NotificationManagers**

Um also mit dem Benutzer in Kontakt zu treten, haben wir folgende Methoden kennen gelernt:

Dialoge	Wenn eine direkte Benutzerinteraktion nötig ist, z.B. bei Nachfragen. Aber auch wenn ein Fortschritt innerhalb der Activity angezeigt werden soll.
Toast-Notifications	Meldungen, die in direkter Folge einer Benutzerinteraktion ausgelöst werden, z.B. »Die Daten wurden gelöscht«. Toasts sind sinnvoll, wenn der Benutzer präsent ist, die Meldung selbst aber nur informativ ist und nicht bestätigt werden muss.
StatusBar-Notifications	Meldungen, die irgendwann, meist durch Hintergrundprozesse, auftreten und den Benutzer zwar informieren, aber nicht unterbrechen sollen. Sind überall dort einzusetzen, wo es auch nicht sicher ist, dass der Benutzer gerade anwesend ist, und die Nachricht aufgehoben werden soll, bis der Benutzer darauf reagieren kann oder will.

Tabelle 3.35: **Benachrichtigungsarten**

3.11.9 Styles und Themes

Ein wichtiges Thema, um unseren Anwendungen ein bestimmtes Look&Feel zu geben, sind die Styles und Themes.

Ein Theme ist einfach eine Sammlung von Styles, die so gut wie alle Aspekte der Oberfläche betreffen: Hintergründe, Schriftgrößen, -farben und -stile, bestimmte Maßangaben etc.

Die Angaben eines Themes werden als Vorgabewerte bei der Instanziierung von Widgets benutzt. Welche Attribute ein Widget haben kann, ist innerhalb der Attribut-Ressourcen definiert. Für das Android-Framework finden sich diese Attributinformationen im Web unter:

`http://android.git.kernel.org/?p=platform/frameworks/base.git;a=blob_plain;f=core/res/res/values/attrs.xml;hb=HEAD`

die Theme-Definitionen unter

`http://android.git.kernel.org/?p=platform/frameworks/base.git;a=blob_plain;f=core/res/res/values/themes.xml;hb=HEAD`

und die Style-Definitionen unter

`http://android.git.kernel.org/?p=platform/frameworks/base.git;a=blob_plain;f=core/res/res/values/styles.xml;hb=HEAD`

Wir können das gesamte Repository von Android unter der Adresse `http://android.git.kernel.org` ansehen und studieren.

Wenn wir die Datei `attrs.xml` genauer anschauen, finden wir für alle Klassen, die wir mittels Layout-Resource erzeugen können, `<declare-stylable ...>`-Elemente, in denen alle Attribute der Klasse aufgeführt sind. Wenn nun ein Widget erzeugt wird, dann werden all diese Attribute mit den Werten aus der `themes.xml` vorbelegt, wenn sie dort deklariert sind. Dadurch erhalten alle Widgets ihr Aussehen abhängig vom zugewiesenen Theme.

Am Beispiel der TextView wird das Ganze deutlich. In der attrs.xml ist folgende Deklaration zu finden:

```
<attr name="textColor" format="reference|color" />
```

Damit wird ein Attribut mit Namen `textColor` innerhalb des Frameworks definiert, das entweder eine Farbe oder aber eine Referenz auf eine Stilressource sein kann.

Später finden wir die Deklaration:

```
<attr name="textAppearance"  format="reference"/>
```

Hier wird das Attribut mit Namen `textAppearance` als Referenz auf eine Stilressource deklariert. Es gibt innerhalb des TextView-Widgets ein Element vom Typ `TextAppearance`, das die Farbe, Größe und weitere Angaben für das Aussehen des Textes beinhaltet. Für dieses Element gibt es ebenfalls eine Deklaration in der `attrs.xml`, die alle diese Attribute aufzählt.

```
<declare-stylable name="TextAppearance">
    [...]
    <attr name="textColor"/>
    <attr name="typeface"/>
    [...]
</declare-stylable>
```

Und auch die TextView besitzt eine Deklaration, die unter anderem das Attribut textAppearance einführt.

```
<declare-stylable name="TextView">
    [...]
    <attr name="textColor"/>
    <attr name="textAppearance"/>
    [...]
</declare-stylable>
```

Innerhalb der themes.xml findet sich für das Standard-Theme folgender Eintrag:

```
<style name="Theme">
    [...]
    <item name="textViewStyle">@android:style/Widget.TextView</item>
    [...]
</style>
```

Hier finden wir also ein Attribut textViewStyle, das wiederum auf eine Style-Ressource verweist, die wir dann in styles.xml finden:

```
<style name="Widget.TextView">
    <item name="android:textAppearance">?android:attr/textAppearanceSmall</
item>
</style>
```

Der Style Widget.TextView also setzt für das Attribut textAppearance (das ja durch die attrs.xml als Element von TextView eingeführt wurde) wiederum eine Referenz auf ein Attribut mit dem Namen textAppearanceSmall, das dann schlussendlich wiederum in der themes.xml definiert wurde:

```
<item name="textAppearanceSmall">@android:style/TextAppearance.Small</item>
```

Hier finden wir wieder eine Referenz auf einen Stil, der dann endlich in der styles.xml auch konkrete Werte aufweist:

```
<style name="TextAppearance.Small">
    <item name="android:textSize">14sp</item>
    <item name="android:textStyle">normal</item>
    <item name="android:textColor">?textColorSecondary</item>
</style>
```

Wenn wir nun eine TextView in einer Layout-Ressource erstellen, wird ihr Aussehen wie folgt festgelegt: Für alle Attribute einer TextView (deklariert in attrs.xml) werden die aus dem Theme und den Style-Ressourcen aufgelösten Werte eingesetzt. Das Theme besitzt ein Attribut textViewStyle, das auf den Style Widget.TextView zeigt. Dieser Style wiederum bestimmt, dass das Attribut textAppearance des Widgets auf den Inhalt des At-

tributs `textAppearanceSmall` gesetzt werden soll, das wiederum im Theme auf den Style `TextAppearance.Small` zeigt. Dieser Style setzt nun die Attribute des Widgets u.a. auf eine Textgröße von 14 sp. Die Textfarbe ist hier wieder eine Referenz auf ein Attribut innerhalb des Themes, das die eigentliche Textfarbe enthält (und die wiederum als Referenz auf eine Farbressource ausgeführt ist).

Wenn wir das Standardaussehen eines Widgets ändern wollen, dann können wir einzelne Attribute überschreiben. Flexibler ist es allerdings, dem Widget einen eigenen Stil zuzuweisen, der z.B. von einem Standardstil erbt und unsere Änderungen beinhaltet.

Wie können wir das aus obiger Betrachtung ableiten? Im Prinzip ist das sehr einfach. Wir müssen lediglich eine eigene `styles.xml` erstellen und dort einen Stil anlegen, der das Aussehen unseres Widgets bestimmt:

```
<style name="MeineTextViewTextAppearance">
    <item name="android:textSize">28sp</item>
    <item name="android:textColor">@color/meinetextview_color</item>
</style>
```

Innerhalb des Layouts können wir diesen Style referenzieren:

```
<TextView
android:id="@+id/meine_textview"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:textAppearance="@style/MeineTextViewTextAppearance"
/>
```

Damit setzen wir für diese `TextView` unsere Definition des Textoutfits.

Wenn wir aber wollen, dass alle unsere `TextViews` so aussehen, dann müssen wir ein eigenes Theme erstellen und den Wert des Attributs `textViewStyle` überschreiben:

```
<style name="MeinTheme" parent="android:Theme.Holo.Light">
    <item name="android:textViewStyle">@style/MeineTextView</item>
</style>
```

Und wir müssen unsere `styles.xml` noch ergänzen:

```
<style name="MeineTextView">
    <item name="android:textAppearance">@style/MeineTextViewTextAppearance</
item>
    <item name="android:gravity">center</item>
</style>
```

Das reicht allerdings noch nicht ganz. Wir müssen unserer Anwendung auch noch mitteilen, dass sie mit unserem Theme arbeiten soll:

```
<application android:name="..." android:theme="@style/MeinTheme" [...] >
```

oder für eine einzelne Activity:

```
<activity android:name="..." android:theme="@style/MeinTheme" [...] >
```

Voilà. Wenn wir nun eine `TextView` erstellen, wird unser Theme ausgewertet und im Endeffekt unsere Version des Textstils benutzt. Und durch die Deklaration von `parent="android:Theme.Holo.Light"` stellen wir gleichzeitig sicher, dass unser Theme vom Android-Standardtheme, bzw. in diesem Falle von der »lichten Variante« des Android 3 Themes (Theme.Holo), erbt und alle anderen Widgets noch so aussehen wie gewünscht.

Wenn wir allerdings nur ausgewählte TextViews »stylen« wollen, benutzen wir am besten das style-Attribut:

```
<TextView
style="MeineTextViewTextAppearance"
android:id="@+id/meinetextview"
[...]
/>
```

Damit können wir Widgets ganz selektiv mit einem eigenen Stil formatieren.

Wann benutzen wir nun welche der dargestellten Techniken? Grundlage für eine eigene Gestaltung ist immer die Deklaration eines Styles. Diesen Style können wir mit dem `style`-Attribut für alle Widgets setzen, die so aussehen sollen, wie der Style das vorgibt. Wenn wir wollen, dass alle Widgets eines bestimmten Typs als Standard immer diesen Style haben, dann müssen wir ein eigenes Theme verwenden und den Style als Vorgabe für das Widget setzen.

Wenn wir nun einen Stil für eines unserer Widgets aus dem Theme benutzen, dann wissen wir dass das Widget genauso aussieht wie alle anderen Widgets, die diesen Stil (und dieses Theme) benutzen.

Ein Theme wird entweder auf Anwendungsebene oder auf Activity-Ebene zugeordnet, und alle Widgets innerhalb der Anwendung bzw. der Activity greifen auf die definierten Stile des Themes zu.

Ein Style wiederum ist eine Sammlung von Eigenschaften und konkreten Eigenschaftswerten, die das Aussehen eines Widgets bestimmen. Der Style erhält einen Namen und zählt die Attribute und deren Werte auf.

Der Vorteil, einen Style zu verwenden statt ein Widget direkt mit den Attributen und Werten zu konfigurieren, liegt auf der Hand: Wollen wir das Aussehen eines bestimmten Widgettyps ändern, können wir die Änderung an der Style-Ressource vornehmen und alle Widgets, die diesen Style verwenden, erfahren diese Änderung.

Eine wichtige Sache bei der Anwendung von Stilen auf Widgets ist, dass Widgets auch verschiedene Zustände annehmen können. Eingabefelder z.B. können den Fokus erhalten, Buttons können gedrückt oder losgelassen sein. Abhängig vom aktuellen Status des Widgets sollen die Widgets ggf. auch anders aussehen. Ein gedrückter Button soll auch wie ein gedrückter Knopf gezeichnet werden. Zu diesem Zweck müssen Farben und Hintergründe für die unterschiedlichen Status angegeben werden. Wenn wir die Attribute eines `EditText`-Widgets betrachten, finden wir aber keine speziellen Attribute für die verschiedenen Status. Um die verschiedenen Status zu reflektieren, gibt es aber sogenannte Color-Statelist-Ressourcen und Statelist-Drawable-Ressourcen. Innerhalb dieser Ressourcen können Angaben für die unterschiedlichen Status gemacht werden:

android:state_pressed	Bei "true" wird die Ressource angewendet, wenn das Widget gedrückt ist, wenn z.B. ein Knopf gedrückt oder ein Klick darauf ausgelöst wird. Bei "false" wird die Ressource angewendet, wenn das Widget nicht gedrückt ist.
android:state_focused	Bei "true" wird die Ressource angewendet, wenn das Widget den Fokus hat, z.B. wenn ein Button über die Tastatur, den Trackball oder das D-Pad angesteuert wurde oder wenn in ein Eingabefeld geklickt wird. Bei "false" wird die Ressource angewendet, wenn das Widget den Fokus nicht hat.
android:state_selected	Bei "true" wird die Ressource angewendet, wenn das Widget ausgewählt ist, z.B. wenn ein Tab geöffnet wurde oder ein Listenelement ausgewählt wurde. Bei "false" wird die Ressource angewendet, wenn das Widget nicht ausgewählt ist.
android:state_checkable	Bei "true" wird die Ressource angewendet, wenn das Widget angekreuzt werden kann. Bei "false" wird die Ressource angewendet, wenn das Widget nicht angekreuzt werden kann. **Hier ist die grundsätzliche Eigenschaft, nämlich ankreuzbar zu sein, des Widgets gemeint, nicht ob das Widget aktiv oder inaktiv ist. Das wird über android:state_enabled angezeigt.** Eine CheckBox z.B. kann (grundsätzlich) immer angekreuzt werden, ein Menüeintrag kann per Eigenschaft ankreuzbar gemacht werden.

Tabelle 3.36: **Die verschiedenen Status eines Widgets**

android:state_checked	Bei "true" wird die Ressource angewendet, wenn das Widget angekreuzt wurde. Bei "false" wird die Ressource angewendet, wenn das Widget nicht angekreuzt wurde.
android:state_enabled	Bei "true" wird die Ressource angewendet, wenn das Widget aktiv ist und Ereignisse empfangen kann. Bei "false" wird die Ressource angewendet, wenn das Widget nicht aktiv ist. Eine übliche Darstellungsweise ist, das Widget ausgegraut darzustellen.
android:state_window_ focused	Bei "true" wird die Ressource angewendet, wenn das Fenster (die Activity oder das Fragment), zu dem das Widget gehört, den Fokus hat. Bei "false" wird die Ressource angewendet, wenn das Fenster den Fokus nicht hat.

Tabelle 3.36: **Die verschiedenen Status eines Widgets (Forts.)**

Die Klasse `View` und ihre Ableitungen liefern mit der Methode onCreateDrawableState(...) den aktuellen Status der View zurück, damit das View-System abhängig vom Status die richtigen Ressourcen für Farbe und die Darstellung herausfinden und anwenden kann. Wenn wir ein eigenes Widget schreiben, dann müssen wir den Status unseres Widgets, sofern dieser nicht von der Basisklasse verwaltet wird, selbst verwalten und zurückliefern:

Listing 3.67: **Beispiel für die Statusverwaltung aus der Klasse CompoundButton.java**

```
private static final int[] CHECKED_STATE_SET = {
    R.attr.state_checked
};
[…]
@Override
protected int[] onCreateDrawableState(int extraSpace) {
    final int[] drawableState = super.onCreateDrawableState(extraSpace + 1);
    if (isChecked()) {
        mergeDrawableStates(drawableState, CHECKED_STATE_SET);
    }
    return drawableState;
}
```

Die in der Tabelle aufgeführten Attribute für die Ressourcen finden ihre Entsprechung in der `R.attr`-Struktur (hier: `R.attr.state_checked`), und damit lassen sich die Status durch `mergeDrawableStates(drawableState, CHECKED_STATE_SET)` mit den durch die Basisklasse verwalteten Status zusammenführen.

INFO

In der Regel müssen wir uns darum nicht kümmern, denn Android liefert ja einen ganzen Strauß von Widgets, die wir als Basis für unsere eigenen Widgets nehmen können und die sich bereits um die Statusverwaltung kümmern. Für uns ist aber der technische Hintergrund interessant, falls wir einmal eine neue, tolle Komponente bauen wollen für die es noch keine entsprechende Basis gibt.

Anwendung finden die `ColorStateList`- und `StateListDrawable`-Ressourcen in den meisten Fällen zum Styling von eigenen Eingabefeldern und Knöpfen.

Die Ressourcen werden in eigenen XML-Dateien angelegt, der Dateiname bildet wie bei anderen Ressourcen auch die Ressourcen-ID. `ColorStateList`-Ressourcen werden im Verzeichnis `res/color` abgelegt, die `StateListDrawable`-Ressourcen unter `res/drawable`.

Listing 3.68: **mybutton_color.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true" android:color="#ffff0000"/>
    <item android:state_focused="true" android:color="#ff0000ff"/>
    <item android:color="#ff000000"/> <!-- Standardfarbe! -->
</selector>
```

In diesem Beispiel definieren wir unterschiedliche Farben für die Status `state_pressed`, `state_focused` und die Standardfarbe, wenn der Button weder gedrückt noch irgendwie sonst in einen bestimmten Zustand versetzt wurde.

ACHTUNG

Wenn bei der Darstellung die Ressource ausgewertet wird, dann wird von oben nach unten nach dem ersten passenden Eintrag gesucht. Wir dürfen die Standardfarbe also niemals an erste Stelle oder mitten rein setzen, da ansonsten die nächsten Einträge gar nicht mehr beachtet werden. Ebenso ist die Reihenfolge `state_pressed` und `state_focused` entscheidend, da das Drücken Vorrang vor dem Fokus haben muss. Drehte man die Reihenfolge um, dann wird ein Button, der den Fokus hat, nie die Farbe beim Drücken wechseln.

Um einen Button nun mit dieser Ressource zu belegen, setzen wir das Attribut `android:textColor` entsprechend auf diese Ressource:

Listing 3.69: **Auszug aus dem Layout mit Zuweisung der Ressource**

```
[…]
<Button
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:textColor="@color/mybutton_color"
android:text="Button 1"/>
[…]
```

Wie wir bereits gelernt haben, ist es aber geschickter, dem Button einen Style zuzuweisen oder aber, um alle unsere Buttons zu verändern, das Theme zu ergänzen. Das hat den Vorteil, dass wir später ganz einfach das Aussehen des Buttons mit `StateListDrawable`-Ressourcen bestimmen können.

Listing 3.70: **Auszug aus der styles.xml**

```
<style name="MeinButtonStyle" parent="android:Widget.Button">
    <item name="android:textColor">@color/mybutton_color</item>
    <item name="android:background">@drawable/mybutton</item>
</style>
```

Listing 3.71: **Auszug aus der themes.xml**

```
<style parent="android:Theme.Holo.Light" name="MeinTheme">
    <item name="android:textSize">28sp</item>
    <item name="android:textViewStyle">@style/MeineTextView</item>
    <item name="android:buttonStyle">@style/MeinButtonStyle</item>
</style>
```

Der Button selbst benötigt dann keine weitere Stilangabe:

Listing 3.72: **Der Button, wenn ein Theme verwendet wird**

```
[…]
<Button
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:text="Button 1"/>
[…]
```

Bei der Anlage des eigenen Button-Styles ist wiederum die Angabe des Parents wichtig, von der unser Stil erst mal erbt: `parent="android:Widget.Button"`. Wenn wir das nicht machen, dann sehen wir zunächst gar nichts, weil die Textgröße etc. dann natürlich nicht definiert ist.

Für den Hintergrund des Buttons kommt nun auch das 9-Patch-Drawable zum Einsatz. Mit dem 9-Patch können wir, wie bereits beschrieben, Bilder erzeugen die sich quasi verlustlos ausdehnen lassen.

Listing 3.73: **Definition der StateListDrawable in mybutton.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<selector
  xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true" android:drawable="@drawable/mybut➥
      ton_pressed" />
    <item android:drawable="@drawable/mybutton_normal" /> <!--  Standard!
-->
</selector>
```

Die Referenz `android:drawable="@drawable/mybutton_normal"` verweist hier auf eben jenes 9-Patch-Drawable mit Namen `mybutton_normal.9.png`, in dem wir das Aussehen des normalen Buttons definieren:

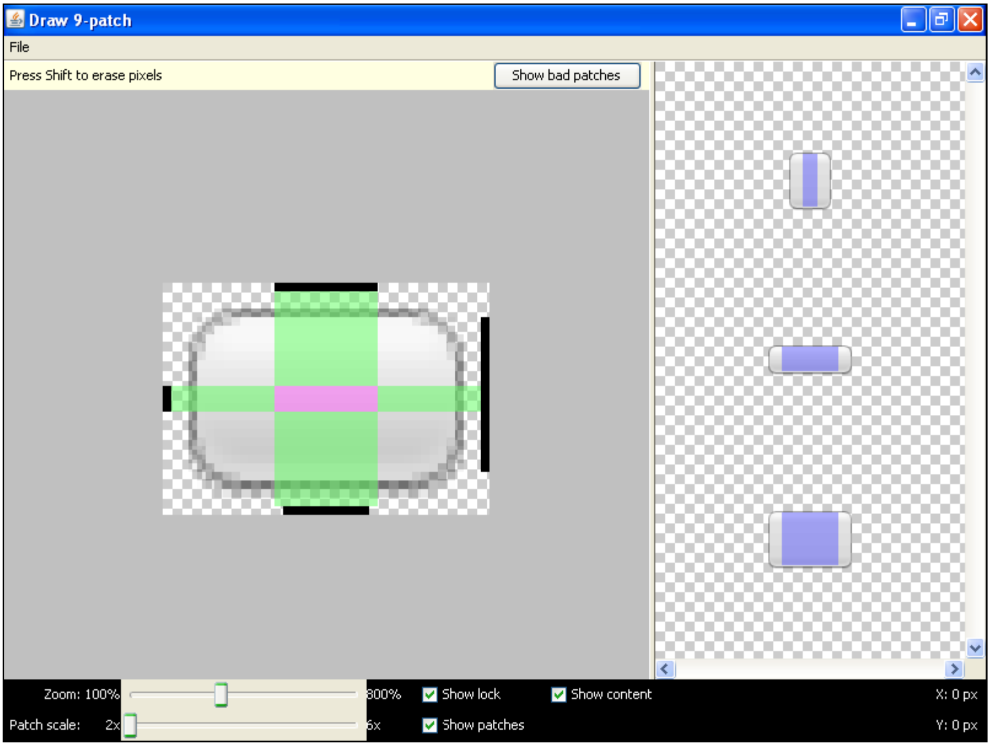


Abbildung 3.26: **Erstellen der Ausdehnungsbereich für den Button**

Den Button selbst zeichnen wir in einem uns genehmen Grafikprogramm, im Draw 9-Patch-Werkzeug, das mit dem SDK mitgeliefert wird, bestimmen wir die Ausdehnungsflächen.

TIPP

Wenn wir eigene zusammengesetzte Widgets auf Basis einer ViewGroup bauen, dann ist das Attribut `android:addStatesFromChildren="true"` interessant. Damit teilen wir der Gruppe, die wiederum Widgets beinhaltet, mit, dass ihr Zustand aus den Zuständen der enthaltenen Elemente errechnet wird. Damit kann z.B. eine Art Formularfeld erstellt werden, dass aus einer `TextView` und einem `EditText` besteht, die Gruppe selbst aber wie ein Eingabefeld mit dessen Hintergrund, Umrandung und Textfarbe formatiert wird. Das haben wir bereits im Abschnitt über das Erstellen eigener Widgets benutzt.

Über die Styles und Themes lassen sich bestehende Widgets mannigfaltig verändern und darstellen, ohne dass wir viel programmieren müssen. In Verbindung mit zusammengesetzten Widgets dienen die Styles und Themes aber auch dazu, uns das Styling eigener Widgets zu erleichtern.

TIPP

Nutzen wir also die Styles und Themes konsequent und verzichten möglichst immer auf die Angabe einzelner Attribute in den Layout-Ressourcen.

3.11.10 Die Action Bar im Detail

Die Action Bar stellt in der Tablet-Umgebung ein zentrales Bedienelement für die Anwendungen dar. Hier können wir für unsere Activities übergreifende Elemente unterbringen:

1. Das Icon der Applikation bzw. der Activity
2. Den Titel der Applikation bzw. der Activity
3. Tabs als Navigationselement z.B. für Fragmente
4. Drop-down Boxen als Navigationselement
5. Eigene Widgets, z.B. für eine Suche
6. Action Items aus dem Optionenmenü der Activity

Das Icon in der Action Bar kann selbst als Action Item benutzt werden und als »Home«-Button oder auch als »Up«-Button dienen:

Listing 3.74: **Das Icon als Home-Button nutzen**

```
[…]
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId())
    {
    case android.R.id.home:
        Intent intent = new Intent(this, Spielwiese3.class);
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
        startActivity(intent);
        return true;
    default:
        Toast.makeText(this, "Ausgewählt: "+item.getTitle(), Toast.LENGTH_
LONG).show();
    }
    return super.onOptionsItemSelected(item);
}
[…]
```

Das Antippen des Icons wird an den `onOptionsItemSelected(…)`-Handler delegiert und mit der ID `android.R.id.home` signalisiert. Entscheidend zur Realisierung ist das Flag `Intent.FLAG_ACTIVITY_CLEAR_TOP` beim Starten der Home-Activity. Dadurch wird keine neue Instanz der Activity gestartet, sondern es werden alle Activities auf dem Stack abgebaut, bis unsere Home-Activity wieder erreicht und »on Top« ist.

INFO

Ohne das Flag würde eine neue Instanz auf den Stack gelegt und der Druck auf »Back« ließe uns wieder zur vorherigen Activity zurückkehren.

Statt das Icon als »Home«-Button zu verwenden, können wir das Icon auch als »Up«-Button darstellen, z.B. um aus der Detailansicht einer Notiz, einer E-Mail oder eines Bildes wieder zur Übersicht zurückzukehren.

Um die Action Bar entsprechend zu manipulieren, können wir innerhalb einer Activity per `getActionBar()` auf die Action Bar zugreifen:

Listing 3.75: **Action Bar Icon zum »Up«-Button umfunktionieren**

```
@Override
public void onStart()
{
    super.onStart();
    ActionBar actionBar = getActionBar();
    actionBar.setDisplayHomeAsUpEnabled(true);
}
```

Die ID, die an den Handler übergeben wird, ist aber in beiden Fällen die gleiche. Es kommt auf uns an, die Mimik zwischen »Home« und »Up« korrekt zu implementieren:

Listing 3.76: **Implementierung des »Up«-Buttons durch Aufruf von finish()**

```
case android.R.id.home:

ActionBar actionBar = getActionBar();
if ((actionBar.getDisplayOptions() & ActionBar.DISPLAY_HOME_AS_UP) == Ac➥
    tionBar.DISPLAY_HOME_AS_UP)
{
    finish();
}
else
{
    Intent intent = new Intent(this, Spielwiese3.class);
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
    startActivity(intent);
}
return true;
```

Die Navigation als Tabs oder auch als Drop-Down Box können benutzt werden um schnell zwischen Ansichten innerhalb der Anwendung zu wechseln, z.B. zwischen verschiedenen Einstellungsdialogen zu wechseln oder die Sortierung der Anzeige umzuschalten.

```
private CharSequence[] getNavigationItems()
{
    return this.getResources().getTextArray(R.array.actionbar_navigation_
items);
}
```

Hier laden wir die Bezeichnungen für die Tabs bzw. die Listeneinträge aus einem String-Array.

```
private void initList() {
    ActionBar actionBar = getActionBar();
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
    SpinnerAdapter mSpinnerAdapter = ArrayAdapter.createFromResource(this,
R.array.actionbar_navigation_items,
```

```
    android.R.layout.simple_spinner_dropdown_item);
    actionBar.setListNavigationCallbacks(mSpinnerAdapter, this);
}
```

Entscheidend ist die Umschaltung des Modus mit `setNavigationMode(…)`. Hier setzen wir den Modus auf `NAVIGATION_MODE_LIST`. Die Einträge werden mit einem `SpinnerAdapter` verbunden, der Adapter wird mit `setListNavigationCallbacks(mSpinnerAdapter,this)` an die Liste gebunden. Hier bestimmen wir die Activity selbst als Empfänger der Auswahl eines Eintrags.

Listing 3.77: **Navigation per Tab oder per Liste**

```
private void initTabs() {
    ActionBar actionBar = getActionBar();
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    CharSequence[] items = getNavigationItems();
    for (int i=0; i<items.length; i++)
    {
        actionBar.addTab(actionBar.newTab().setText(items[i]).
setTabListener(this));
    }
}
```

Hier werden die Einträge als Tabs mittels `newTab()` hinzuggefügt, und jedes einzelne Tab erhält unsere Activity als `TabListener` zugeordnet, um auf die Auswahl oder Abwahl eines Tabs zu reagieren.

Damit unsere Activity auf die Auswahl reagieren kann, implementieren wir die entsprechenden Handler:

Listing 3.78: **Implementieren der Handler**

```
public class ActionBarActivity extends Activity implements TabListener,
OnNavigationListener {
[…]
@Override
public void onTabReselected(Tab tab, FragmentTransaction ft) {
}

@Override
public void onTabSelected(Tab tab, FragmentTransaction ft) {
    Toast.makeText(this, tab.getText(), Toast.LENGTH_SHORT).show();
}
@Override
public void onTabUnselected(Tab tab, FragmentTransaction ft) {
}
@Override
public boolean onNavigationItemSelected(int pos, long id) {
    CharSequence[] items = getNavigationItems();
    Toast.makeText(this, items[pos], Toast.LENGTH_SHORT).show();
    return true;
}
[…]
}
```

In Verbindung mit Fragmenten kann die Action Bar auch noch die Spur unserer Navigation per Bread Crumbs verfolgen. Diese Technik wird im Abschnitt über Fragmente näher erläutert.

In Verbindung mit Action Items, die eigentlich ganz normale Menüeinträge des Optionenmenüs der Activitiy (oder des Fragments) sind, können auch Widgets in die Action Bar eingebunden werden. Damit kann z.B. eine Suchbox bereitgestellt werden. Wenn nicht genügend Platz ist, dann kann die Suche weiterhin über den zugehörigen Menüeintrag geöffnet werden.

Listing 3.79: **Zugriff auf die Action View**

```
<item
android:actionViewClass="android.widget.SearchView"
android:id="@+id/menu_search"
android:showAsAction="ifRoom"
android:title="@string/searchlabel"
android:icon="@android:drawable/ic_menu_search"/>
```

Listing 3.80: **Auszug aus actionbarmenue.xml – Deklarieren einer Action View**

Um die Action View innerhalb unserer Activity zu nutzen, z.B. zur Konfiguration der View, greifen wir innerhalb von `onCreateOptionsMenu(…)` auf die View zu:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.actionbarmenue, menu);
    SearchManager searchManager = (SearchManager) getSystemService(Context.
SEARCH_SERVICE);
    SearchView searchView = (SearchView) menu.findItem(R.id.menu_search).
getActionView();
    searchView.setIconifiedByDefault(false);
    searchView.setSearchableInfo(searchManager.getSearchableInfo(getComponen
tName()));
    return true;
}
```

In diesem Beispiel konfigurieren wir die `SearchView`. Wie die Suche konfiguriert und benutzt wird betrachten wir im Zusammenhang mit den Content -Providern, denn die Suche ergibt ja erst in Verbindung mit gespeicherten Daten wirklich Sinn.

Alternativ zu der Angabe einer Klasse können wir die Action View auch durch eine Layout-Ressource definieren:

Listing 3.81: **Deklaration eines Layouts für die Action View**

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<item android:id="@+id/menu_search"
android:title="Search"
android:icon="@drawable/ic_menu_search"
```

```
android:showAsAction="ifRoom"
android:actionLayout="@layout/searchview" />
</menu>
```

Damit sind wir extrem flexibel, was das Bestücken der Action Bar angeht. Menüeinträge werden einfach mittels MenüRessourcen eingefügt:

Listing 3.82: **Action Bar-Menü**

```
<?xml version="1.0" encoding="utf-8"?>
<menu
xmlns:android="http://schemas.android.com/apk/res/android">
<item
    android:title="@string/menueItemOption1"
    android:id="@+id/option1"
    android:icon="@drawable/icon"
    android:showAsAction="ifRoom|withText"></item>
<item
    android:title="@string/menuItemOption2" android:id="@+id/option2"></
item>
<item
    android:id="@+id/option3"
    android:title="@string/menuItemOption3">
    <menu>
        <item android:id="@+id/option31" android:title="@string/menuItemOp➥
           tion31"></item>
        <item android:id="@+id/option32" android:title="@string/menuItemOp➥
           tion32"></item>
    </menu>
</item>
<item
    android:actionViewClass="android.widget.SearchView"
    android:id="@+id/menu_search"
    android:showAsAction="ifRoom"
    android:title="@string/searchlabel"
    android:icon="@android:drawable/ic_menu_search"/>
</menu>
```

Entscheidend für die Darstellung ist das Attribut `android:showAsAction="…"`:

ifRoom	Eintrag wird nur angezeigt wenn genug Platz ist. Wenn er in der Leiste angezeigt wird, dann ist er im Menü selbst nicht mehr vorhanden.
never	Eintrag wird nie in der Leiste angezeigt.
withText	Zeigt zusätzlich zum Icon den Text des Eintrags in der Leiste an, ansonsten nicht.
always	Der Eintrag wird immer in der Leiste angezeigt. Sinnvoll z.B. für Action Views wie ein Suchfeld, das immer sichtbar bleiben soll.

Tabelle 3.37: **Optionen für showAsAction**

Die Action Bar kann auch mittels Styles und Themes ganz individuell angepasst werden. Dazu bietet das Framework folgende Style-Referenzen an, die in einem eigenen Theme benutzt werden können:

android:actionBarTabStyle	Styling der Tabs
android:actionBarTabBarStyle	Styling der gesamten TabBar
android:actionBarTabTextStyle	Styling des Textes innerhalb der Tabs
android:actionDropDownStyle	Styling der Drop-down-Liste
android:actionButtonStyle	Styling der Buttons in der Leiste

Tabelle 3.38: **Style-Referenten der Action Bar**

Damit können wir die Action Bar ganz individuell anpassen. Wie wir Themes und Styles benutzen, haben wir bereits besprochen, so könnte eine entsprechende Definition für die Action Bar aussehen:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CustomActionBar" parent="android:style/Theme.Holo.Light">
        <item name="android:actionBarTabTextStyle">@style/customActionBar➥
           TabTextStyle</item>
        <item name="android:actionBarTabStyle">@style/customActionBarTab➥
           Style</item>
        <item name="android:actionBarTabBarStyle">@style/customActionBar➥
           TabBarStyle</item>
    </style>
```

Der obere Abschnitt definiert das Theme als Abwandlung von Theme.Holo.Light. Innerhalb der Abwandlung werden die Style-Referenzen deklariert, die im Folgenden die jeweiligen Attribute definieren.

Listing 3.83: **Customizing der Action Bar**

```
    <style name="customActionBarTabTextStyle">
        <item name="android:textColor">#2966c2</item>
        <item name="android:textSize">20sp</item>
        <item name="android:typeface">sans</item>
    </style>
    <style name="customActionBarTabStyle">
        <item name="android:background">@drawable/actionbar_tab_bg</item>
        <item name="android:paddingLeft">20dp</item>
        <item name="android:paddingRight">20dp</item>
    </style>
    <style name="customActionBarTabBarStyle">
        <item name="android:background">@drawable/actionbar_tab_bar</item>
    </style>
</resources>
```

3.11.11 Datenbindung an Views

Wie bekommen wir nun aber Daten in unsere Views, wie das z.B. der Kontaktmanager oder Kalenderanwendungen tun? Zum einen müssen die Daten ja irgendwo gespeichert werden. Das soll hier noch nicht zum Thema werden, dazu schauen wir uns später die *Content-Provider* an.

Android führt eine sehr hohe Abstraktion zur Datenbindung an Views ein und nennt dieses Konzept *Adapter*. Das Adapterkonzept folgt dem MVC- (Model-View-Controller-) Muster, bei dem die Datenhaltung (im Businessmodel) von der Anzeige (in den Views) bzw. dem Zugriff über den Controller entkoppelt ist. Die Views bzw. die Zugriffe werden über eine standardisierte Schnittstelle im Controller, hier und im Folgenden Adapter genannt, abgewickelt, und der eigentliche Zugriff in der Implementierung des Adapters durchgeführt. Das heißt, dass es für unterschiedliche Datenquellen unterschiedliche Adapter gibt, die aber nach »oben« hin eine einheitliche Zugriffsschnittstelle bieten.

Die Basisschnittstelle in Android, von der alle Adapter abstammen, ist das Interface `Adapter`.

Alle Ableitungen, die irgendwie Daten an die Views liefern wollen/sollen/müssen/können, müssen das Interface `Adapter` implementieren. Die allermeisten Adapter erben aber einfach von der Basisklasse `BaseAdapter`, die eine Grundlage für Adapter liefert, die sowohl in listenförmigen Views als auch in sogenannten Spinnern eingesetzt werden können. Die listenförmigen Views, zu denen `ListViews`, aber auch `Grids` gehören, zeigen mehrere Einträge auf einmal und erlauben das Durchrollen durch die Einträge. Ein `Spinner` zeigt nur einen Eintrag, lässt aber aus allen Einträgen denjenigen auswählen, der angezeigt werden soll. Spinner sind unter Windows oft als Comboboxen ausgeführt, bei denen man die Einträge durch Aufklappen sichtbar macht und dann auswählt. Es gibt aber auch Widgets, die das mit Pfeiltasten realisieren, um durch die Einträge nach oben oder nach unten durchzuschalten.

Das Interface Adapter liefert nun also nach oben eine definierte Schnittstelle. Die wichtigsten Methoden sind dabei:

1. getCount()
2. getItem(int position)

Mittels `getCount()` wird die Anzahl an Einträgen ermittelt, und mit `getItem(position)` kann ein Eintrag geholt werden. Wie effizient jetzt die Implementierung arbeitet, entscheidet über Performace und Speicherverbrauch des Adapters.

Es gibt einige Adapter, die auf einfachen internen Daten wie Listen oder Arrays operieren. Es ist offensichtlich, dass diese Adapter nur für eine begrenzte Zahl an Einträgen verwendet werden sollten und nicht dazu dienen, große Datenbestände zu repräsentieren, denn dann müssten die Daten ja komplett in den Speicher geladen werden.

Aber dennoch liefern diese Adapter schnelle Erfolge, z.B. um ein Auswahlmenü als `ListView` zu realisieren, wie wir das in der Spielwiese gemacht haben. Dort werden die Menüein-

träge in eine `List` von `Maps` eingetragen, und diese Liste kann mit dem `SimpleAdapter` direkt an eine `ListView` gebunden werden.

Effizientere Adapter sind Cursor-Adapter, die auf einem Cursor-Konstrukt operieren. Cursor sind ein Konzept von Datenbankmanagementsystemen, ein Cursor dient hier zum Durchwandern von Datenmengen, der Cursor zeigt dabei immer auf genau eine Zeile in der Datenmenge. Die Dateien einer Zeile der Datenmenge sind in Spalten organisiert. Der Kontaktmanager z.B. operiert auf einer Datenbank, in der eine Zeile den Kontakt repräsentiert und aus den Spalten Name, Vorname und weiteren Attributen aufgebaut ist.

Da der Cursor durch die Datenmenge bewegt, oft vor- und rückwärts, und in manchen Systemen sogar frei positioniert werden kann, ohne dass der ganze Datenbestand in den Hauptspeicher geladen werden muss, stellt diese Form der Datenorganisation und des Zugriffs eine sehr effiziente Möglichkeit dar, auch große Datenbestände performant und speicherschonend darzustellen.

Der Adapter, mit dem man wohl am meisten arbeiten wird ist, der `SimpleCursorAdapter`. Äußerlich gleicht er dem `SimpleAdapter`, er wird im Gegensatz zu diesem allerdings nicht an eine Liste gebunden, sondern an einen `Cursor`.

Woher dieser Cursor kommt, ist dann erstmal nicht relevant, wir müssen nur wissen, wie die Daten aussehen, die der Cursor liefert (womit die Herkunft natürlich schon eine gewisse Relevanz erhält). Mit dem Konzept der Content-Provider stellt Android nun aber ein sehr offenes und allgemeingültiges System zur Datenspeicherung bereit. Die Speicherung und Verwaltung der Daten geschieht in den Content-Providern (diese legen Daten z.B. in der SQLite, ggf. aber auch in einem *Cloud-Service* ab). Alle Content-Provider verfügen wiederum über eine standardisierte Schnittstelle, um einen Cursor auf ihren Datenbestand zu erhalten (wenn die Provider das erlauben).

Hier findet sich das Konzept der Modularisierung und losen Kopplung des Android Systems wieder, über das wir schon des Öfteren philosophiert haben, und zwar auf der Ebene der Datenspeicherung und des Datenzugriffs.

Allen Adaptern ist gemein, dass man angibt, welche Daten wo in unserer View erscheinen sollen, das ist das sogenannte Mapping.

Einfache Adapter

Einfache Adapter arbeiten mit Arrays von Objekten oder mit Listen, die eine Map – ein assoziatives Array – enthalten können.

Das Array transportiert in jeder Zeile des Arrays genau ein Objekt. Dieses Objekt wird dann in der Regel als Zeichenkette interpretiert und in der View dargestellt. Ein Array kann Objekte eines Typs enthalten oder auch Objekte unterschiedlichen Typs, wobei wir die Umwandlung in eine Zeichenkette durch das Überschreiben der Methode `toString()` der Objekte durchführen.

TIPP

Die Zeilen eines Arrays transportieren in der Regel immer nur einen Wert, und nicht mehrere Spalten. Wenn wir in einer Zeile mehrere Spalten darstellen wollen, benutzen wir eine Liste von assoziativen Arrays.

Ein Beispiel für ein Array wäre ein Array von Zeichenketten, das die Wochentage repräsentiert:

Listing 3.84: **Array direkt im Programm initialisieren**

```
String[] wochentage = { "Sonntag", "Montag", "Dienstag", "Mittwoch", "Do-
nerstag", "Freitag", "Samstag" };
```

Alternativ, in diesem Fall auch besser, wenn wir die Applikation später übersetzen wollen, kann das Array aus einer Ressource erzeugt werden:

Listing 3.85: **Array aus Resource laden**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string-array name="wochentage_array">
        <item>Sonntag</item>
        <item>Montag</item>
        <item>Dienstag</item>
        <item>Mittwoch</item>
        <item>Donnerstag</item>
        <item>Freitag</item>
        <item>Samstag</item>
</string-array>
</resources>
…
String[] wochentage = getResources().getStringArray(R.array.wochentage_ar➥
    ray);
```

Um jetzt die Wochentage an eine ListView zu binden, benutzen wir den ArrayAdapter. In diesem Beispiel benutzen wir eine ListActivity, die die komplette Funktionalität für das Darstellen einer Liste zur Verfügung stellt.

Listing 3.86: **Verwenden des Array-Adapters**

```
public class PickWeekDayActivity extends ListActivity {

protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);

    String[] wochentage = getResources().getStringArray(R.array.wochenta➥
       ge_array);

    ArrayAdapter<String> wochentageAdapter = new ArrayAdapter<String>(this,a
ndroid.R.layout.simple_list_item_1,android.R.id.text1,wochentage);
    setListAdapter(wochentageAdapter);
}

}
```

Hier benutzen wir im ArrayAdapter Ressourcen, die Android bereits zur Verfügung stellt. `android.R.layout.simple_list_item_1` referenziert die Layout-Ressource (im Grunde eine XML-Date im `layout`-Verzeichnis), die für das Aussehen einer Zeile zuständig ist. `android.R.id.text1` referenziert in dieser Ressource das Element, im Standardlayout eine `TextView`, die den Text der Zeile aufnimmt.

Damit wissen wir auch schon, wie wir das Aussehen einer einzelnen Zeile einer `ListView` beeinflussen können. Statt die eingebauten Layouts und Ressourcen-IDs zu verwenden, können wir selbst ein Layout zur Verfügung stellen und im Adapter benutzen.

Wie man sieht, kann der Array-Adapter tatsächlich nur eine Spalte darstellen. Wenn wir pro Zeile mehrere Spalten darstellen wollen müssen wir uns den `SimpleAdapter` anschauen.

TIPP

In diesem Beispiel benutzen wir ein Array von Strings als Datenquelle. Der Array-Adapter kann aber auch eine Liste von Strings bzw. eine Liste von Objekten als Datenquelle verwenden:

```
List<String> wochentage = new LinkedList<String>();
wochentage.add("Sonntag");
ArrayAdapter<String> wochentageAdapter = new ArrayAdapter<String>(this,➥
android.R.layout.simple_list_item_1,android.R.id.text1,wochentage);
setListAdapter(wochentageAdapter);
```

Nehmen wir an, wir wollen ein Auswahlmenü mit einer ListView erstellen und zu jedem Eintrag eine kurze Beschreibung darstellen, dann benötigen wir als Datenquelle eine Liste, deren Einträge aus dem Titel und der Beschreibung bestehen. Eine solche Datenstruktur bauen wir z.B. mit einer Liste auf, die ein assoziatives Array pro Zeile transportiert.

Ein assoziatives Array ist ein Array, dessen Indexwerte nicht ganze Zahlen sein müssen, sondern beliebige Schlüsselwerte, somit auch Strings sein können.

Listing 3.87: **Verwendung des SimpleAdapter**

```
public class SimpleAdapterListViewActivity extends ListActivity {
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);

    List< Map<String,String>> menueListe = new LinkedList<
Map<String,String> >();

    Map<String,String> menueEintrag1 = new HashMap<String,String>();
    menueEintrag1.put("Titel", "Erster Eintrag");
    menueEintrag1.put("Beschreibung", "Beschreibung des ersten Eintrags");

    menueListe.add( menueEintrag1 );

    Map<String,String> menueEintrag2 = new HashMap<String,String>();
    menueEintrag2.put("Titel", "Zweiter Eintrag");
    menueEintrag2.put("Beschreibung", "Beschreibung des zweiten Eintrags");

    menueListe.add( menueEintrag2 );
```

```
        SimpleAdapter menueAdapter =
        new SimpleAdapter(this,
         menueListe,
         android.R.layout.two_line_list_item,
         new String[] { "Titel", "Beschreibung" },
         new int[] { android.R.id.text1,android.R.id.text2 }
        );

        setListAdapter(menueAdapter);
    }
    }
```

Wichtig ist hier das folgende Konstrukt:

```
SimpleAdapter menueAdapter =
new SimpleAdapter(this,
 menueListe,
 android.R.layout.two_line_list_item,
 new String[] { "Titel", "Beschreibung" },
 new int[] { android.R.id.text1,android.R.id.text2 }
);
```

Wie beim Array-Adapter legen wir auch hier das Aussehen der einzelnen Zeilen fest, diesmal wählen wir das Standardlayout `android.R.layout.two_line_list_item`. Dieses Layout stellt zwei Zeilen (zwei TextViews) zur Verfügung, wobei die erste Zeile fett gesetzt wird und die zweite in normaler Schriftdicke, also genau das Richtige für einen Titel und eine kurze Beschreibung.

Da nun pro Zeile mehrere Spalten angezeigt werden können, müssen wir hier angeben, welche Spalte aus der Liste auf welche TextView des Layouts abgebildet wird:

```
new String[] { "Titel", "Beschreibung" },
new int[] { android.R.id.text1,android.R.id.text2 }
```

Die erste Zeile ist das »Von«, die zweite Zeile beschreibt das »Nach«. Wir erkennen, dass das »Von« die jeweiligen Indizes bzw. Schlüssel der Listenzeilen darstellt:

```
Map<String,String> menueEintrag2 = new HashMap<String,String>();
menueEintrag2.put("Titel", "Zweiter Eintrag");
menueEintrag2.put("Beschreibung", "Beschreibung des zweiten Eintrags");
menueListe.add( menueEintrag2 );
```

Das »Nach« wiederum ist ein Array von Ressourcen-IDs, die die jeweilige TextView innerhalb des Layouts referenzieren.

TIPP

Diese Konstruktion wird auch bei den Cursor-Adaptern verwendet, die dem SimpleAdapter im Prinzip recht ähnlich sind.

Genau wie bei dem Array-Adapter können wir das Layout unserer ListView also auch selbst bestimmen und auch mehr als zwei Spalten darstellen, wenn wir entsprechende Widgets im Layout bereitstellen.

Wie man leicht erkennen kann, ist die Bereitstellung von mehrspaltigen Daten in dieser Form nicht besonders effizient, wenn wir größere Datenmengen anzeigen wollen.

Um größere Datenmengen, z.B. aus einer SQL-Datenbank, anzeigen zu können, verwenden wir die Cursor-Adapter.

Cursor-Adapter

Wie wir schon besprochen haben, liefert Android eine SQLite-Datenbank im Betriebssystem mit. Das ist insofern bemerkenswert, da SQL mit seiner standardisierten Abfragesprache einen strukturierten **und** performanten Zugriff auf Daten bietet und uns Android damit von der Last befreit, für unsere Datenbanken ein eigenes Ablage- und Suchsystem zu entwerfen. Konsequent nutzen auch die meisten Anwendungen, so auch die mitgelieferten Anwendungen, die SQLite-Datenbank, um Daten wie Kontakte, Telefonnummern, Lesezeichen, Playlists etc. zu speichern.

TIPP

Sobald wir strukturierte Daten in unserer Anwendung verwalten wollen, sollten wir ebenfalls möglichst ausschließlich die Speicherung in der Datenbank vorsehen und nicht damit anfangen, eigene Dateistrukturen aufzubauen.

Der Zugang zu den Daten von Anwendungen wird über die Content-Provider abgewickelt. Das Sicherheitssystem von Android schottet die Anwendungen voneinander ab, so dass der direkte Zugriff auf Dateien und/oder Datenbanken nicht möglich ist. Hier müssen zum einen die bereitgestellten Schnittstellen verwendet und zum anderen die Erlaubnis angefordert werden, auf die Daten anderer Anwendungen zuzugreifen.

Der Zugriff auf einen speziellen Content-Provider wird mittels des sogenannten Content-URI durchgeführt. Der Content-URI ist mit einer Webadresse vergleichbar. Die Webadresse adressiert eindeutig eine Ressource im World-Wide-Web, der Content-URI adressiert eindeutig einen Content-Provider auf unserem Gerät.

Über den Content-URI fordern wir in unserer Anwendung einen Cursor an, der den Zugriff auf die einzelnen Datensätze der Anwendung, z.B. der Kontakte, erlaubt. Dieser Cursor wiederum wird mittels des Cursor-Adapters mit unseren Widgets verbunden, damit wir die Daten sehen können.

Wo bekommen wir den Content-URI nun her?

Wenn eine Anwendung, auch unsere eigene, Daten verwaltet, implementiert sie einen entsprechenden Content-Provider, und mit diesem bestimmt sie auch den Content-URI der enthaltenen Daten. Ein Content-URI hat in der Regel die Form content://<AUTHORITY>/<TABLE_NAME>.

Durch das Schema `content` qualifizieren wie den URI als Content-URI. Eine Webadresse z.B. hat das Schema `http`, damit wird bestimmt, dass die Daten über das http-Protokoll ausgetauscht werden, das Schema `content` bestimmt hier, dass auf die Daten über einen Content-Provider zugegriffen wird.

`<AUTHORITY>` bezeichnet eindeutig den Besitzer der Daten, also die Anwendung, die die natürliche Autorität über die Daten hat. Um eine eindeutige Benennung zu erreichen, wird hier in der Regel der Name des Anwendungspakets und der Content-Provider-Klasse benutzt:

```
content://de.androidpraxis.spielwiese.notizenprovider
```

`<TABLE_NAME>` bestimmt dann noch die Tabelle, in der die Daten verwaltet werden. In der Regel benutzt eine Anwendung mehrere Tabellen um die Daten zu speichern:

```
content://de.androidpraxis.spielwiese.notizenprovider/notiz
```

```
content://de.androidpraxis.spielwiese.notizenprovider/kategorie
```

Die Anwendungen definieren per Konvention die Konstante `CONTENT_URI` für die jeweiligen Content-Provider und Tabellen.

Die mit Android mitgelieferten Content-Provider sind z.B. innerhalb des Packages `android.provider` organisiert. Darin befinden sich z.B. die Kontakte in der Klasse `ContactsContract`, wiederum unterteilt in verschiedene Kategorien bzw. Tabellen.

Die Kontaktverwaltung innerhalb des Android-Systems stellt im Prinzip eine sehr anspruchsvolle Implementierung eines Content-Providers dar. Die Kontaktverwaltung bietet nämlich die Möglichkeit, zum einen Kontakte aus verschiedenen Quellen (Google Mail, Facebook, Exchange Server etc.) zu vereinheitlichen, zum anderen stellt sie die Möglichkeit bereit, Kontaktdaten um beliebige Zusatzinformationen (Bild, erste, zweite, n-te Telefonnummer, Ereignisse, Lieblingsessen etc.) zu erweitern.

Der Aufhänger für jeden Kontakt ist dann auch ein ziemlich allgemeines Objekt `ContactsContract.Data`, das nun den Basis-URI `ContactsContract.Data.CONTENT_URI` definiert, über den wir auf die Kontaktdaten zugreifen können.

Im Detail beschäftigen wir uns bei der Betrachtung der Content-Provider damit, jetzt wollen wir einfach mal per Cursor-Adapter auf Daten zugreifen.

```
public class ShowContactsActivity extends ListActivity {
private static String[] PROJECTION = new String[] {
    ContactsContract.Data._ID, android.provider.ContactsContract.CommonData➥
       Kinds.StructuredName.DISPLAY_NAME
};
```

Mittels der Struktur `PROJECTION` definieren wir die Spalten die wir aus der Datenbank lesen wollen. Hier sind das die `ID` der Kontakte und der Anzeigename (`DISPLAY_NAME`) des Kontakts.

ACHTUNG

Die Angabe der ID ist in jedem Zugriff auf einen Content-Provider obligatorisch, besonders dann, wenn die Ergebnisse in einem Adapter verwendet werden. Über die ID wird jeder Datensatz eindeutig identifiziert, und innerhalb der Adapter wird über diese ID auf die Datensätze zugegriffen.

Wie schon gesagt, stellt der Content-Provider der Kontaktverwaltung eine anspruchsvolle Implementierung dar. Eigentlich hat die Tabelle gar keine speziellen Felder für Namen oder Telefonnummern, sondern nur generische Felder (`DATA1` bis `DATA15`). Die eigentliche Bedeutung dieser Felder bestimmt sich aus dem Typ der Datenzeile (`CONTENT_ITEM_TYPE`, siehe weiter unten). Wir wollen hier die Datenzeilen der Kontakte selektieren, die den Namen eines Kontakts darstellen, daher benutzen wir die Spalte

```
android.provider.ContactsContract.CommonDataKinds.StructuredName.DISPLAY_
NAME
```

und selektieren aus dem Content-Provider dann die Zeilen für den Typ

```
android.provider.ContactsContract.CommonDataKinds.StructuredName.CONTENT_
ITEM_TYPE
```

```
private static String[] DISPLAY = new String[] {
android.provider.ContactsContract.CommonDataKinds.StructuredName.DISPLAY_
NAME
};
```

Die Struktur `DISPLAY` definiert die Spalten, die wir aus dem Content-Provider anzeigen wollen. In diesem Beispiel wollen wir nur die Spalte `DISPLAY_NAME` anzeigen. Wenn wir mehrere Felder anzeigen wollen, so hat das Auswirkungen auf die Struktur `PROJECTION` und die Struktur `DISPLAY`.

```
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    Cursor cur = managedQuery(ContactsContract.Data.CONTENT_URI, PROJECTION,
ContactsContract.Data.MIMETYPE +" = ?", new String[] { android.provider.Con➥
    tactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE }, andro➥
    id.provider.ContactsContract.CommonDataKinds.StructuredName.DISPLAY_
NAME);
```

Diese Zeile ist die erste Magie zum Zugriff auf den Content-Provider. Per `managedQuery (ContactsContract.Data.CONTENT_URI, …)` wird der Zugriff auf den Content-Provider hinter ContactsContract.Data angefordert.

Der Parameter PROJECTION beschreibt die Spalten, die selektiert werden sollen. Die beiden Parameter

```
ContactsContract.Data.MIMETYPE +" = ?"
```

und

```
new String[] { android.provider.ContactsContract.CommonDataKinds.Structured➥
    Name.CONTENT_ITEM_TYPE }
```

gehören zusammen. Ersterer bestimmt die Abfrage an den Content-Provider, wir wollen nur Zeilen eines bestimmten `MIMETYPE` (Inhaltstyps) haben. Das Fragezeichen ist der Platzhalter, dessen Wert durch den nächsten Parameter bestimmt wird. Wir wollen ja die Namen der Kontakte selektieren und fordern die Daten zu

```
android.provider.ContactsContract.CommonDataKinds.StructuredName.CONTENT_
ITEM_TYPE
```

an.

TIPP

Man sieht im Zusammenhang mit den SQL-Abfragen häufig Statements, die einfach durch das Zusammenhängen von Zeichenketten erzeugt werden. Das ist besonders bei der Abfrage ziemlich gefährlich, bietet das doch das Einfallstor für sogenannte SQL-Injections. Wir sollten bei der Formulierung von Abfragen immer Parameter benutzen und niemals die Abfrage durch die Verkettung von Variablen aufbauen.

```
    SimpleCursorAdapter cursorAdapter = new SimpleCursorAdapter(this,
    android.R.layout.simple_list_item_1,
    cur,
    DISPLAY,
    new int[] {android.R.id.text1});
```

Hier wird nun der Cursor mit einem Adapter verbunden. Wie schon bei den einfachen Adaptern bestimmen wir hier das **von** und das **nach**: Von `DISPLAY` nach `new int[] {android.R.id.text1}`. Es ist einfach zu sehen, dass wir sowohl die Struktur PROJECTION als auch DISPLAY als auch den letzten Parameter erweitern oder ändern müssen, wenn wir mehrere Spalten anzeigen wollen.

Listing 3.88: **Benutzen des SimpleCursorAdapters**

```
    setListAdapter(cursorAdapter);
}
}
```

Und damit wird der Adapter mit der List-View verbunden.

Wichtig ist jetzt noch, dass wir im Manifest festlegen dass wir auf den Content-Provider zugreifen möchten:

Listing 3.89: **Bekanntmachen, dass wir lesenden Zugriff auf die Kontakte benötigen**

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="de.androidpraxis.Spielwiese"
android:versionCode="1"
android:versionName="1.0">
[…]
<uses-permission android:name="android.permission.READ_CONTACTS"></uses➥
    permission>
[…]
</manifest>
```

Wie wir sehen, ist das Binden von Daten an die Benutzeroberfläche über die Adapter recht einfach zu bewerkstelligen. Die Herausforderung ist, besonders beim Zugriff auf die Content-Provider, die Datenstruktur zu verstehen und den Zugriff entsprechend durchzuführen.

ACHTUNG

In diesem Beispiel benutzen wir das Konstrukt der `managedQuery(…)`. Das ist bis API-Level 10 auch die übliche Vorgehensweise. Ab API-Level 11 bietet Android aber das *Loader-Framework*, mit dem der Datenzugriff auf große Datenmengen sehr einfach in den Hintergrund verlagert werden kann und zukünftig der `managedQuery(…)` vorzuziehen ist.

Das Loader-Framework betrachten wir später genauer.

In den bisherigen Beispielen haben wir Listenansichten benutzt. Ein wichtiges Widget ist auch der Spinner. Der Spinner zeigt immer einen ausgewählten Eintrag aus einer Liste von Einträgen. Eine Drop-down-Box ist ein Beispiel für einen Spinner.

Spinner setzen wir überall da ein, wo der Benutzer aus einer relativ geringen Anzahl von Einträgen einen Wert auswählt, z.B. um eine Notiz, ein Bild oder eine Adresse mit einer Kategorie zu versehen. In Datenbankanwendungen nennt man das auch Nachschlagelisten.

Der Spinner wird ähnlich der ListViews mittels eines Adapters mit Daten versorgt, wobei die Standard-Ressourcen für die Darstellung der Items etwas anders sind und zusätzlich noch die Ressource für die Drop-down-Items angegeben werden müssen:

Listing 3.90: **Verwenden eines Spinners, um die verfügbaren Alben anzuzeigen**

```
cursorAdapterAlbums = new SimpleCursorAdapter(getContext(),
    android.R.layout.simple_spinner_item,
    null,
    new String[] { MediaStore.Images.Media.BUCKET_DISPLAY_NAME } ,
    new int[] {android.R.id.text1});
getLoaderManager().initLoader(0, null, this);
Spinner s2 = (Spinner) findViewById(R.id.media_album);
cursorAdapterAlbums.setDropDownViewResource(android.R.layout.simple_spinner_
dropdown_item);
s2.setAdapter(cursorAdapterAlbums);
```

Dem Beispiel liegt folgendes Layout zugrunde:

```
[…]
<Spinner
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:id="@+id/media_album" />
[…]
```

Standardmäßig möchte Android 3 wohl die Spinner als Popup-Fenster ausführen, zumindest wenn sie in Dialogen ausgeführt werden. Es gibt keine Möglichkeit per Attribut oder Methode die Darstellung umzuschalten. Das lässt sich mit einer eigenen Klasse, die von Spinner abgeleitet wird, jedoch umgehen.

Dazu leiten wir den Spinner folgendermaßen ab und passen die Konstruktoren an:

Listing 3.91: **Umwandeln eines Spinners zu einem Drop-down-Spinner**

```
public static class MySpinner extends Spinner
{
    public MySpinner(Context context, AttributeSet attrs, int defStyle,
    int mode) {
        super(context, attrs, defStyle, mode);
    }
    public MySpinner(Context context, AttributeSet attrs, int defStyle) {
        this(context,attrs,defStyle,Spinner.MODE_DROPDOWN);
    }
    public MySpinner(Context context, AttributeSet attrs) {
        this(context,attrs,android.R.attr.spinnerStyle,Spinner.MODE_DROP➥
    DOWN);
    }
    public MySpinner(Context context, int mode) {
        super(context, mode);
    }
    public MySpinner(Context context) {
        this(context,Spinner.MODE_DROPDOWN);
    }
};
```

Je nach Gusto können wir dann für die einzelnen Items auch ein anderes Layout wählen:

Listing 3.92: **Anderes Layout für die Einträge der Drop-down Liste**

```
[…]
cursorAdapterAlbums.setDropDownViewResource(android.R.layout.simple_drop➥
    down_item_1line/*simple_spinner_dropdown_item*/);
[…]
```

INFO

Der eigene Spinner wird dann in dieser Form im Layout verwendet:

`<de.androidpraxis.SpielwieseLibrary3.widgets.MySpinner …>`

Wenn der Spinner nun für eine Nachschlageliste eingesetzt wird, dann müssen wir natürlich dafür sorgen, dass der entsprechende Datensatz im Spinner selektiert wird, der zum Nachschlagewert unseres primären Datensatzes (z.B. des Bildes oder der Notiz) gehört.

Per Konvention müssen die CursorAdapter mit Daten versorgt werden, die eine stabile ID besitzen. In Verbindung mit den Content-Providern ist das (in den meisten Fällen) der primäre, automatische Schlüssel der Tabelle im Feld _ID. Wenn unser Bild also zu einem Album gehört, dann speichern wir nicht den Namen des Albums ab, sondern genau den Schlüssel, d.h. die ID des Albums. Dann können wir im Adapter des Spinners die Einträge durchgehen und die IDs vergleichen:

```
Spinner s2 = (Spinner) findViewById(R.id.media_album);
long albumID = cursor.getLong(cursor.getColumnIndex("ALBUM_ID"));
for (int i=0; i<cursorAdapterAlbums.getCount(); i++)
{
```

```
    if (cursorAdapterAlbums.getItemId(i)==albumID)
    {
        s2.setSelection(i);
        break;
    }
}
```

Der Weg zurück funktioniert über das Auslesen der ausgewählten ID:

Listing 3.93: **Selektieren und Auslesen des Nachschlagewertes im Spinner**

```
Spinner s2 = (Spinner) findViewById(R.id.media_album);
long albumID = s2.getSelectedItemId();
cursor.setLong(cursor.getColumnIndex("ALBUM_ID"),albumID);
```

ACHTUNG

Dieses Beispiel ist konstruiert und so nicht direkt in der Spielwiese vorhanden, da es im MediaStore das Konzept des Albums in dieser Form nicht gibt. Das Beispiel geht von einer eigenen datenbankgestützten Albenverwaltung aus.

Anpassen der ListView

Android liefert für die Widgets Standardlayouts mit, die für die meisten Anwendungen ausreichen.

ListActivity und ListFragment arbeiten beide mit einem Vorgabelayout für die Listendarstellung , und Android liefert Standardlayouts für die Darstellung der Einträge, die wir mit unseren Adaptern verwenden können:

Listing 3.94: **Standardliste mit einer Textzeile**

```
cursorAdapter = new SimpleCursorAdapter(getActivity(),
android.R.layout.simple_list_item_1,
null,
DISPLAY ,
new int[] { android.R.id.text1 });
```

Die Standardlayouts (wie `android.R.layout.simple_list_item_1`) finden wir in `android.R.layout`. Wichtig ist oft noch das `android.R.layout.two_line_list_item`, um zwei Zeilen in einem Eintrag darzustellen.

Abhängig davon, welches Layout man für den Listeneintrag wählt, haben wir auch verschiedene Views innerhalb dieser Standardlayouts zur Verfügung, um die Daten darzustellen: `new int[] { android.R.id.text1 }` für `android.R.layout.simple_list_item_1`, oder `new int[] { android.R.id.text1, android.R.id.text2 }` für das `android.R.layout.two_line_list_item`.

Damit haben wir schon Vorgaben, die wir in vielen Fällen nutzen können. Aber der Appetit kommt ja bekanntlich beim Essen, und schon bald wollen wir die Einträge komplexer darstellen, z.B. mit einem Vorschaubild und zwei Zeilen Text.

Um das zu erreichen, können wir ein eigenes Layout für die Darstellung der Listeneinträge nutzen:

Listing 3.95: **Eigenes Layout für die Listeneinträge**

```
cursorAdapter = new SimpleCursorAdapter(getActivity(),
R.layout.imageview_listitem,
null,
DISPLAY ,
new int[] {R.id.imageview_title,R.id.imageview_description,R.id.imageview_
image});
[…]
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
style="@style/listViewActivatedStyle"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:textAppearance="?android:attr/textAppearanceMedium"
>
    <ImageView android:layout_height="wrap_content" android:layout_
width="wrap_content" android:src="@drawable/icon" android:id="@+id/image➥
    view_image"></ImageView>
    <LinearLayout android:layout_height="match_parent" android:id="@+id/li➥
    nearLayout1" android:orientation="vertical" android:layout_width="match_
parent">
        <TextView android:layout_height="wrap_content" android:layout_
width="match_parent" android:text="TextView" android:id="@+id/imageview_tit➥
    le"></TextView>
        <TextView android:layout_height="wrap_content" android:layout_
width="match_parent" android:text="TextView" android:id="@+id/imageview_de➥
    scription"></TextView>
    </LinearLayout>
</LinearLayout>
```

Dieses Beispiel wird noch einmal ein wenig komplexer durch das Bild, das in der ImageView dargestellt werden soll. Die Standardbindung der Daten kann keine Bilder verarbeiten, dazu müssen wir einen eigenen Mechanismus an den Adapter anschließen:

Listing 3.96: **Eigene Methode der Datenbindung**

```
cursorAdapter.setViewBinder(new SimpleCursorAdapter.ViewBinder()
{
@Override
public boolean setViewValue(final View view, Cursor cursor, int columnIndex)
{
    if (view.getId() == R.id.imageview_image)
    {
        final ImageView imageView = (ImageView)view;
        final long id = cursor.getLong(0);
        Bitmap bm = imageCache.get(id,imageView);
        if (bm!=null)
        {
        imageView.setImageBitmap(bm);
        }
        else
```

```
        {
        imageView.setImageResource(R.drawable.icon);
        }
        return true;
    }
    return false;
}
});
```

Hier reagieren wir darauf, dass der Adapter die Daten an die Views binden möchte. Wir erhalten in setViewValue(...) nacheinander die IDs unserer Views und den Cursor, aus dem die Daten gebunden werden. Für alle Standard-Views (hier: die Texte) liefern wir einfach false zurück, um die Standardbindung durchzuführen. Im Falle der ID `R.id.imageview_image` allerdings setzen wir das zu den Daten gehörende Bild in die View ein. Hier wird das durch einen Cache-Mechanismus erledigt, der in der Spielwiese detailliert zu studieren ist.

TIPP

Der ViewBinder ist ein mächtiger Mechanismus, um die Daten so in der View darzustellen, wie wir es möchten, z.B. lassen sich hier dann auch berechnete Felder darstellen, die so in der View gar nicht vorhanden sind oder aber Formatierungen durchführen, z.B. um Währungsbeträge oder Datumswerte darzustellen.

Wenn wir noch einen Schritt weiter gehen wollen, und die komplette ListView für unsere Activity oder das Fragment zu gestalten können wir auch ein komplett eigenes Layout per setContentView(...) zuweisen:

Listing 3.97: **Eigenes Layout für die ListView in layout/my_list_layout.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="vertical"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:paddingLeft="8dp"
 android:paddingRight="8dp">

     <ListView android:id="@android:id/list"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     android:background="#00FF00"
     android:layout_weight="1"
     android:drawSelectorOnTop="false"/>

     <TextView android:id="@android:id/empty"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     android:background="#FF0000"
     android:text="No data"/>
</LinearLayout>
```

Wichtig sind die IDs der Elemente, die genau wie hier im Listing angegeben werden müssen. Über die IDs kann die ListActivity die korrekte Anbindung an unser Layout vornehmen.

Listing 3.98: **Setzen des eigenen Layouts in MyListActivity**

```
class MyListActivity extends ListActivity
{
[…]
@Override
public void onCreate(Bundle savedInstanceState)
{
[…]
    setContentView(R.id.my_list_layout);
[…]
}
```

INFO

Dieses Beispiel ist so in der Spielwiese nicht zu finden.

Wir kennen aus einigen Anwendungen, dass die Meldung »Keine Daten gefunden« oder »Keine Einträge vorhanden« angezeigt wird, wenn eine Datenquelle keine Daten liefert, sei es, weil noch keine angelegt sind oder weil eine Suche keine Ergebnisse gebracht hat.

Im obigen Beispiel wird die View, die in diesem Fall angezeigt werden soll, innerhalb des Layouts mit der ID `@android:id/empty` deklariert.

Wir können das aber auch ohne ein eigenes Layout benutzen, in dem wir die View programmtechnisch in der Activity erzeugen:

Listing 3.99: **Setzen der View für die Anzeige, wenn keine Daten verfügbar sind**

```
TextView emptyView = new TextView(getActivity());
emptyView.setGravity(Gravity.CENTER_HORIZONTAL);
emptyView.setText("Keine Einträge gefunden");
ViewParent parent = getListView().getParent();
if (parent instanceof FrameLayout)
{
    ((FrameLayout)parent).addView(emptyView, new FrameLayout.
LayoutParams(LayoutParams.MATCH_PARENT,LayoutParams.MATCH_PARENT));
}
getListView().setEmptyView(emptyView);
```

Wichtig ist hier das Zufügen der View zum Vater- oder Mutterelement der ListView.

TIPP

Die View kann natürlich auch wieder aus einer XML-Ressource erstellt werden und weitere Elemente beinhalten. Denkbar wäre ein Button, der es erlaubt, neue Einträge zu erstellen wenn keine Daten vorhanden sind.

3.11.12 Drag&Drop

Durch den Platzgewinn, den ein Tablet ab 7" Bildschirmdiagonale und mit einer höheren Auflösung als ein Smartphone bringt, bietet sich die Implementierung des Drag&Drop-Musters für Anwendungen an, um neue Bedienmöglichkeiten zu erschließen.

Drag&Drop kennt man durchaus schon von den Homescreens, wenn wir Anwendungsverknüpfungen oder App-Widgets platzieren und verschieben. Dieser Mechanismus war allerdings bis Android 3 kein offizieller Bestandteil der API und wurde nicht standardisiert bereitgestellt.

Ab Android 3 ist das Drag&Drop im View-System integriert und erlaubt uns relativ einfach die Implementierung eigener Drag&Drop-Operationen. In einer Album-Anwendung könnte das Drag&Drop zum Umorganisieren der Medien genutzt oder Listeneinträge könnten per Drag&Drop umsortiert oder in andere Kategorien verschoben werden.

Beim Drag&Drop haben wir immer eine Quelle, ein Widget das per `startDrag(…)` einen Drag&Drop-Vorgang initiiert, ein Datenobjekt, das die Daten enthält, und möglicherweise Listener, die auf Drag-Operationen reagieren.

Die Drag-Listener entscheiden anhand des Datenobjekts, ob sie die Daten empfangen können oder wollen, und kümmern sich darum, die Daten dann auch abzuholen, wenn sie abgelegt werden.

Beim Initiieren des Drag&Drop-Vorgangs, wenn das Datenobjekt erstellt wird, wird dem Drag-Vorgang auch ein Drag-Shadow zugewiesen, der ein Abbild der Daten während des Ziehens repräsentieren soll. Dazu implementieren wir einen `DragShadowBuilder`, der dieses Abbild erzeugt und auch das Zeichnen des Abbilds übernimmt.

Um eine ListView zur Quelle einer Drag&Drop-Operation zu machen bietet sich der OnItemLongClickListener an:

Listing 3.100: **Starten einer Drag&Drop-Operation auf einem Listeneintrag**

```
getListView().setOnItemLongClickListener(new OnItemLongClickListener() {
public boolean onItemLongClick(AdapterView<?> av, View v, int pos, long id)
{
    TextView titleView = (TextView)v.findViewById(R.id.imageview_title);
    final String title = (String) titleView.getText();
    ClipData data = ClipData.newUri(ShowImagesFragment.this.getActivity().
getContentResolver(), title, ContentUris.withAppendedId(contentUri, id));
    v.startDrag(data, new MyDragShadowBuilder(v), null, 0);
    return true;
}
});
```

Hier ist entscheidend, dass unsere ListView auf dem MediaStore operiert und wir die Daten einfach als URI des Content-Providers beschreiben können, daher initialisieren wir das Datenobjekt einfach mit dem URI des Eintrags.

Die übergebene View v ist die View, die den angewählten Listeneintrag repräsentiert.

Der DragShadowBuilder nun erzeugt einen rechteckigen Bereich in der Größe der View und lässt beim Zeichnen des Shadows die View sich einfach selbst auf diesen Bereich zeichnen:

Listing 3.101: **DerDrag ShadowBuilder**

```
private class MyDragShadowBuilder extends View.DragShadowBuilder {
private Drawable mShadow;

public MyDragShadowBuilder(View v) {
    super(v);
    mShadow = new ColorDrawable(Color.GRAY);
    mShadow.setCallback(v);
    mShadow.setBounds(0, 0, v.getWidth(), v.getHeight());
}
@Override
public void onDrawShadow(Canvas canvas) {
    super.onDrawShadow(canvas);
    mShadow.draw(canvas);
    getView().draw(canvas);
}
}
```

Damit haben wir die ListView drag-fähig gemacht. Nun gilt es, eine Ziel-View zu realisieren die als Ziel der Operation dienen kann.

Das erreichen wir durch die Implementierung eines OnDragListeners:

Listing 3.102: **Der OnDragListener**

```
imageView.setOnDragListener(new View.OnDragListener() {
public boolean onDrag(View v, DragEvent event) {
switch (event.getAction()) {
    case DragEvent.ACTION_DRAG_ENTERED:
    imageView.setColorFilter(Color.GREEN,PorterDuff.Mode.LIGHTEN);
    imageView.invalidate();
return true;


case DragEvent.ACTION_DRAG_EXITED:
    imageView.clearColorFilter();
    imageView.invalidate();
    return true;


case DragEvent.ACTION_DRAG_STARTED:
    boolean ok = processDragStarted(event);
    if (ok)
    {
        imageView.setColorFilter(Color.BLUE,PorterDuff.Mode.LIGHTEN);
        imageView.invalidate();
    }
    return ok;

case DragEvent.ACTION_DROP:
    imageView.clearColorFilter();
    imageView.invalidate();
    return processDrop(event, imageView);
}
return false;
}
});
```

Die Aktionen, die der Listener auswerten muss, sind folgende:

ACTION_DRAG_STARTED	Eine Drag&Drop-Operation wird gestartet. Wird an alle Listener geschickt. Wenn eine View als Ziel infrage kommt, muss der Listener »true« zurückliefern. Alle weiteren Ereignisse werden nur an die Listener geschickt, die »true« zurückgeliefert haben. Alle Ziele können ggf. ihr Aussehen verändern, um anzuzeigen, dass sie als Ziel zur Verfügung stehen.
ACTION_DRAG_ENTER	Das Drag-Objekt befindet sich innerhalb der View. Die View kann ihr Aussehen verändern, um anzuzeigen, dass nun das Objekt abgelegt werden könnte.
ACTION_DRAG_EXITED	Das Drag-Objekt wurde aus der View herausbewegt.
ACTION_DRAG_LOCATION	Das Drag-Objekt wird innerhalb der View bewegt. Die Position des Zeigers kann erfragt werden.
ACTION_DROP	Das Drag-Objekt wurde abgelegt. Die View muss entsprechend reagieren, die Daten untersuchen und die beabsichtigte Aktion durchführen.
ACTION_DRAG_ENDED	Die Drag&Drop-Operation wird beendet. Dieses Ereignis ist eigentlich für die Quelle von Belang, um herauszufinden, ob das Ablegen erfolgreich war.

Tabelle 3.39: **Drag&Drop Aktionen**

Um festzustellen, ob die View die Daten erhalten kann, prüfen wir in der Regel den Typ der Daten. Dazu untersuchen wir die Beschreibung des Datenobjekts:

Listing 3.103: **Feststellen ob wir die Daten empfangen können**

```
boolean processDragStarted(DragEvent event) {
ClipDescription clipDesc = event.getClipDescription();
if (clipDesc != null) {
    boolean ok = clipDesc.hasMimeType(MediaStore.Images.Media.CONTENT_TYPE)
|| clipDesc.hasMimeType("image/jpeg");
    String mt = clipDesc.getMimeType(0);
    Log.d(Globals.LOG_TAG,mt);
    return ok;
}
return false;
}
```

Zum Empfangen der Daten beim Ablegen müssen wir dann die einzelnen Items aus dem Datenobjekt extrahieren. Ein Datenobjekt während einer Drag&Drop-Operation kann potenziell eine Liste von Daten enthalten, also nicht nur einen Eintrag transportieren:

Listing 3.104: **Extrahieren und Verarbeiten der Daten**

```
boolean processDrop(DragEvent event, ImageView imageView) {
ClipData data = event.getClipData();
if (data != null) {
    if (data.getItemCount() > 0) {
        Item item = data.getItemAt(0);
        Uri contentUri = item.getUri();
        setImageURI(contentUri);
        return true;
    }
}
return false;
}
```

3.12 Activities

3.12.1 Grundlegendes über Activities

Eine Activity repräsentiert genau einen Bildschirm mit einem Benutzerinterface. Das heißt, dass eine Activity für genau eine Sache zuständig ist und dem Benutzer genau diese eine Sache zeigt und ihn genau mit dieser Sache interagieren lässt.

Ein gutes Beispiel ist eine Adressbuchanwendung, aber allgemein auch alle Anwendungen, die irgendwelche Daten verwalten. Alle diese Awendunge haben ein Muster gemeinsam:

1. Übersicht und Suche von Daten (Anzeige und Durchsuchen des Adressbuchs)
2. Anschauen eines Datensatzes (Anzeige einer Adresse)
3. Erstellen eines Datensatzes (Eintragen einer Adresse)
4. Aktualisieren/Bearbeiten eines Datensatzes (Adresse ändern oder ergänzen)
5. Löschen eines Datensatzes (Löschen einer Adresse)

Jede dieser Activities hat ein eigenes Benutzerinterface. Die Übersicht stellt die Einträge in einer Liste oder Tabelle dar und ermöglicht die Suche in dieser Liste. Anschauen eines Datensatzes öffnet eine weitere Activity, die den Datensatz anzeigt und dann ggf. weitere Aktionen ermöglicht. Erstellen eines Datensatzes öffnet eine Activity, in der die einzelnen Felder des Adressbucheintrags gefüllt werden können.

Unsere Anwendung besteht also aus mindestens einer Activity, meist aber aus mehreren Activities. Man kann sich das in etwa so vorstellen, dass für jedes »Fenster«, das wir dem Benutzer präsentieren, eine Activity benötigt wird. Wir verändern das Aussehen einer Activity niemals, um etwas völlig anderes anzuzeigen oder eine weitere Funktion auszuführen. Daraus folgt auch, dass eine Activity genau ein Layout hat. Das Layout der Activity wird mit Views erstellt. Wie der Name schon sagt, sind Views sichtbare Bestandteile der Benutzeroberfläche wie Text, Eingabefelder, Listen, Tabellen etc. Die Layouts können programmiert

werden, allerdings ist es besser, die Layouts als Ressourcen bereitzustellen. Layout-Ressourcen sind reine XML-Dateien, in denen das Layout und die Elemente der Oberfläche in XML definiert werden. Die Ressourcen haben wir uns schon sehr genau betrachtet, hier liegt wie beschrieben auch der Hund begraben, um Applikationen für verschiedene Bildschirmgrößen und -auflösungen vorzubereiten sowie Texte mehrsprachig zu organisieren und Icons und Bilder bereitzustellen.

Wie wir für die fiktive Adressbuchanwendung beschrieben haben, öffnet eine Activity möglicherweise eine andere Activity. Dadurch legt sich diese neue Activity über die aktuelle Activity, wie auf einen Kartenstapel. Das Starten von Activities führen wir nie direkt, sondern über die Intents aus. Intents sind Nachrichten, die über den entsprechenden Systemservice innerhalb der Laufzeitumgebung verschickt werden und auf die Anwendungsbausteine reagieren.

Üblicherweise werden innerhalb von Applikationen Fenster und Programmfunktionen direkt aufgerufen. Warum macht man das hier nicht genauso?

Nun, ein besonderes Merkmal von Android ist ja gerade der modulare Aufbau. Warum soll man nicht in der eigenen Anwendung die Funktionen der installierten Adressbuchanwendung nutzen, um z.B. einen Kontakt für was auch immer auszuwählen? Oder wieso sollte die Adressbuchanwendung nicht auf die installierte Kameraanwendung zugreifen, um ein Foto von jemandem zu schießen, dessen Adresse wir gerade eingegeben haben?

Damit das aber funktioniert, müssen wir Funktionen anderer Anwendungen aufrufen, und wie wir bereits gelernt haben, sind die Anwendungen strikt voneinander getrennt. Das einzige Bindeglied ist die Laufzeitumgebung respektive das Betriebssystem, das alle Anwendungen kontrolliert und mittels des Kontextes die Anwendungen an die Laufzeitumgebung ankoppelt. Damit wir nun Funktionen anderer Anwendungen nutzen können, müssen wir über das Laufzeitsystem eine Nachricht schicken mit der wir z.B. die Kameraanwendung auffordern, uns ein Foto zu schießen und zurückzugeben (natürlich muss uns die Anwendung das erlauben). Und genau das machen wir mit den Intents. Eine Activity wird also über ein Intent gestartet. Das ist ausnahmslos so, das ist der einzige, standardisierte Weg eine Activity zu starten. Und das gilt sowohl innerhalb einer Applikation als auch über Anwendungsgrenzen hinweg.

Bezogen auf unser fiktives Adressbuch bedeutet das, dass die erste Activity unserer Anwendung über die Laufzeitumgebung gestartet wird, und zwar wiederum von einer eigenen Anwendung, dem Launcher bzw. dem Homescreen. Erinnern wir uns: Android ist extrem modular aufgebaut, und was wir nach dem Einschalten sehen ist nichts anderes als eine Anwendung die mit unseren Anwendungen gleichberechtigt läuft. Unsere Start-Activity unserer Anwendung wird also schon durch ein Intent, gestartet das der Launcher erzeugt. Wie das geht betrachten wir später noch genauer.

Wenn wir nun die Liste der Adressbucheinträge sehen und einen Eintrag auswählen, dann startet die Activity mittels eines Intents, dem der ausgewählte Eintrag übergeben wird, die Activity zum Ansehen (oder bearbeiten) des Eintrags.

3.12.2 Die Activity genauer betrachtet

Um eine eigene Activity zu erstellen, leiten wir unsere Activity von der Basisklasse Activity oder einer spezielleren Activity-Klasse wie `ListActivity` ab.

Unsere Activity muss nun dreierlei implementieren:

1. Das Erzeugen der Benutzeroberfläche
2. Den Lebenszyklus
3. Die eigentliche Funktionalität

Die erste Methode ist die onCreate()-Methode, die auch den ersten Schritt im Lebenszyklus der Activity darstellt, es ist sozusagen die (Wieder-) Geburt der Activity. In der `onCreate(… )`-Methode erstellen wir die Benutzeroberfläche und, bei einer Wiedergeburt, stellen den Zustand der Activity zum Zeitpunkt ihres vorübergehenden Lebensendes wieder her.

Listing 3.105: **onCreate()-Beispiel**

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}
```

In diesem Beispiel steckt schon eine ganze Menge Dinge, obwohl es nur so kurz ist. In `savedInstanceState` steht möglicherweise der Zustand der Activity zu einem Zeitpunkt, zu dem sie vom Betriebssystem aus dem Verkehr gezogen wurde. Bundles sind Objekte, in denen wir kleinere Datenportionen speichern können, um sie über Prozessgrenzen hinweg zu transportieren oder aber auch über die Lebensdauer der Applikation kurzzeitig aufzubewahren. Um das korrekte Speichern des Zustandes müssen wir uns aber selbst kümmern, das passiert üblicherweise in der Methode `onSaveInstanceState(…)`.

ACHTUNG

Dieser Mechanismus ist hauptsächlich dafür gedacht, den Zustand der Benutzeroberfläche zu speichern, und sollte niemals dazu verwendet werden, wichtige Daten wie z.B. den gerade bearbeiteten Datensatz zu speichern. Es kann nicht garantiert werden, dass onSaveInstanceState() überhaupt aufgerufen wird. Wenn z.B. der gerade bearbeitete Datensatz gesichert werden soll, so sollte das in onPause() geschehen, denn diese Methode wird sicher immer aufgerufen, bevor unsere Activity den Fokus verliert.

Weiterhin erkennen wir das Erstellen der Benutzeroberfläche. Mittels `setContentView()` wird die Wurzel einer View-Hierarchie gesetzt, die von der Activity angezeigt wird. In diesem Beispiel ist die View-Hierarchie in einer Ressource definiert, die mit dem Identifizierer `R.layout.main` angesprochen wird. Hinter diesem von der Entwicklungsumgebung generierten Identifizierer »versteckt« sich eine XML-Datei im Verzeichnis `res\layout` mit dem Namen `main.xml`, in der die Benutzeroberfläche definiert wird.

In unserer Activity implementieren wir dann die Methoden, die von der Laufzeitumgebung aufgerufen werden um den Lebenszyklus der Activity zu verwalten. Es ist von Vorteil, sich einen Rahmen zu schaffen, den man immer wieder verwenden kann. In dem Zusatzmaterial zu diesem Buch finden wir im Projekt Spielwiese eine `SimpleLifecycleActivity`, die nichts anderes tut, als die Lebenszyklusmethoden zu implementieren und den Aufruf mittels eines Loggers zu protokollieren.

Sich mit dem Lebenszyklus zu beschäftigen ist sehr wichtig, um gute Anwendungen für mobile Geräte zu schreiben. Zum einen können wir z.B. Daten, die gerade bearbeitet werden, vor dem Vergessen retten, wenn die Anwendung pausiert, und potenzieller Kandidat für das Entfernen aus dem Speicher ist, und zum anderen sollten beim Pausieren der Activity mögliche energiefressende Prozesse wie das Abfragen von Sensorwerten oder das Auslesen von GPS-Daten angehalten und später wieder fortgesetzt werden.

Außerdem behandelt Android viele Änderungen an der Konfiguration des Systems dadurch, dass die laufende Activity einfach neu gestartet wird. Das passiert z.B. dann, wenn wir die Bildschirmausrichtung vom Hochformat ins Querformat ändern, das Gerät in eine DockingStation im Auto einstecken, also immer bei Änderungen die sich auf das Layout und die Funktionalität der Activity auswirken können. Auf den ersten Blick etwas merkwürdig, aber ziemlich logisch, wenn man bedenkt, dass für unterschiedliche Bildschirmausrichtungen (hochkant oder längs) und Sprachen ja auch ggf. unterschiedliche Layouts bzw. Texte benutzt werden müssen. Diese müssen aus den Ressourcen neu geladen werden, und das passiert in der `onCreate()`-Methode, wenn `setContentView()` aufgerufen wird. Dadurch, dass Android in so einem Fall die Activity einfach neu startet, ist es ohne großen Programmieraufwand möglich, auf die neuen Konfigurationen zu reagieren, da dies durch das Ressourcensystem der Laufzeitumgebung automatisch passiert.

INFO

Glücklicherweise kümmert sich das Android-Laufzeitsystem beim Neustart einer Activity auch darum, den Zustand des Userinterface zu sichern und wiederherzustellen. Dazu wird für jede View in der View-Hierarchie selbst die Methode `onSaveInstanceState()` aufgerufen, damit die Views ihren Zustand speichern können. Wir müssen den Views nur eine ID mitgeben, was aber sowieso eine gute Idee ist das für alle View-Elemente zu tun. Wir müssen lediglich dafür sorgen, dass der Zustand von Variablen, die wir selbst eingeführt haben, gesichert und wiederhergestellt werden kann.

Folgender Überblick zeigt die wichtigsten Methoden des Lebenszyklus und was man innerhalb dieser Methoden machen sollte.

onCreate()	Wird aufgerufen, wenn die Activity erstellt wird. Das kann beim ersten Start passieren oder nachdem die Activity aus welchen Gründen auch immer zerstört wurde und der Benutzer z.B. mittels BACK Key hierher zurückkehrt. Wird ebenfalls aufgerufen wenn die Activity durch Konfigurationsänderungen neu gestartet werden muss.
onStart()	Wird aufgerufen, **bevor** die Activity sichtbar wird.
onResume()	Wird aufgerufen, wenn die Activity sichtbar ist und Eingabeereignisse verarbeiten kann. Direkt nach onResume() ist die Activity also im laufenden Zustand und der Benutzer kann mit der Activity interagieren. Hier sollten dann Dinge fortgesetzt werden, die beim Pausieren der Activity gestoppt wurden, z.B. das Horchen auf Sensorwertänderungen, Animationen o.Ä.
onPause()	Wird aufgerufen, **bevor** eine **andere** Activity fortgesetzt oder gestartet wird. Hier sollten Daten gespeichert werden, die noch nicht gespeichert sind, und Prozesse angehalten werden, die Rechenzeit und Energie verbrauchen, z.B. sollte das Horchen nach Sensorwerten hier gestoppt werden. Das sollte allerdings alles sehr schnell passieren, denn die nächste Activity startet erst, wenn diese Methode ausgeführt wurde.
onStop()	Wird aufgerufen, wenn die Activity schlussendlich **nicht mehr** sichtbar ist, z.B. weil sie aus dem Speicher entfernt wurde oder, was meistens der Fall sein dürfte, sich eine **andere** Activity über diese Activity gelegt hat.
onDestroy()	Wird aufgerufen, bevor die Activity tatsächlich aus dem Speicher entfernt wird. Das passiert entweder, wenn der Benutzer die Activity beendet, die Activity entfernt werden muss, um Ressourcen freizumachen, oder die Activity aufgrund von Konfigurationsänderungen neu gestartet werden muss.
onSave-Instance-State()	Wird aufgerufen, **bevor** die Activity zerstört wird. Hier können wir den aktuellen Zustand der Activity speichern um den Zustand bei der Rückkehr zur Activity wiederherzustellen, was z.B. wie beschrieben bei der Änderung der Bildschirmausrichtung passiert. Es gibt keine Garantie dafür, dass onSaveInstanceState() tatsächlich aufgerufen wird. Wird die Anwendung regulär beendet, dann wird diese Methode nicht aufgerufen, und auch unter gewissen extremen Umständen kann die Activity ohne Aufruf der Methoden onStop(), onDestroy() und onSaveInstanceState() zerstört werden. Das bedeutet natürlich, dass man hier keine wirklich wichtigen Dinge speichern sollte, sondern nur die Dinge, die zum Wiederherstellen des Activity-Zustands benötigt werden, wenn der Benutzer sich normal durch den Activity-Stapel arbeitet. **Achtung**: Diese Methode ist nicht dafür gedacht große Datenmengen zu speichern. Einerseits ist das Bundle-Objekt dafür nicht geeignet, andererseits darf diese Methode nicht extrem lange dauern da ansonsten das System bis zur Ausführung der nächsten Activity »hängt«.

Tabelle 3.40: **Übersicht über den Activity-Lebenszyklus**

onRetainNon-Configuration-Instance()	Diese Methode wird aufgerufen, wenn die Activity wegen einer Konfigurationsänderung neu gestartet wird, und dient dazu, ggf. größere Datenmengen hinüberzuretten, um diese Daten nicht neu laden zu müssen. Das ist nützlich bei allen Applikationen, bei denen man gerne öfter zwischen dem Hochformat und Querformat wechselt, z.B. bei einer Bildergalerie. Um nicht jedes Mal die Galerie neu laden zu müssen, kann der aktuelle Zustand der Galerie hier hinübergerettet werden. In onCreate() kann dann mit getLastNonConfigurationState() der Zustand wieder ausgelesen werden.

Tabelle 3.40: **Übersicht über den Activity-Lebenszyklus (Forts.)**

Mittels der `SimpleLifecycleActivity` bzw. der Log-Ausgaben des `MarbleGame` (beides auf der CD zu finden) lässt sich sehr schön der Lebenszyklus unserer Anwendung nachvollziehen.

Hier einige Auszüge, um den Lebenszyklus zu verdeutlichen:

1. Starten der Activity über den Launcher/den Homescreen

Listing 3.106: **Lebenszyklus Log-File 1**

```
02-21 10:52:06.564: DEBUG/de.androidpraxis.marblegame.MarbleGame(1066):
de.androidpraxis.marblegame.MarbleGame.onCreate()
02-21 10:52:06.614: DEBUG/de.androidpraxis.marblegame.MarbleGame(1066):
de.androidpraxis.marblegame.MarbleGame.onStart()
02-21 10:52:06.614: DEBUG/de.androidpraxis.marblegame.MarbleGame(1066):
de.androidpraxis.marblegame.MarbleGame.onResume()
[BACK-Taste wird betätigt]
02-21 10:52:14.765: DEBUG/de.androidpraxis.marblegame.MarbleGame(1066):
de.androidpraxis.marblegame.MarbleGame.onPause()
02-21 10:52:14.924: DEBUG/de.androidpraxis.marblegame.MarbleGame(1066):
de.androidpraxis.marblegame.MarbleGame.onStop()
02-21 10:52:14.934: DEBUG/de.androidpraxis.marblegame.MarbleGame(1066):
de.androidpraxis.marblegame.MarbleGame.onDestroy()
```

Hier kann man sehr schön den Standardablauf erkennen, wenn die Activity gestartet und per BACK-Taste verlassen wird. Wir sehen das `onDestroy()` aufgerufen wird, die Anwendung ist also tatsächlich beendet.

2. Starten über den Launcher und betätigen der HOME-Taste

Listing 3.107: **Lebenszyklus Log-File 2**

```
02-21 10:53:04.774: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onCreate()
02-21 10:53:04.824: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onStart()
02-21 10:53:04.824: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onResume()
[HOME-Taste wird betätigt]
02-21 10:53:11.804: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onPause()
02-21 10:53:12.004: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onStop()
```

Durch das Betätigen der Home-Taste legt sich eine neue Activity (der Homescreen) über unsere Activity. Der Lebenszyklus läuft bis zur onStop()-Methode. Unsere Applikation ist immer noch da, aber inaktiv. Der Homescreen selbst ist so eingerichtet dass man bei Betätigen der HOME-Taste immer an den Ausgangspunkt zurückkehrt. Der Homescreen legt sich also nicht neu auf den Stapel, sondern es wird ein neuer Stapel begonnen. Daher kann man nun mit der BACK-Taste nicht zu unserer Anwendung zurückkehren.

3. Anwendung wieder über den Launcher starten und mit BACK beenden

Listing 3.108: **Lebenszyklus Log-File 3**

```
02-21 10:55:57.275: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onRestart()
02-21 10:55:57.275: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onStart()
02-21 10:55:57.275: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onResume()
[BACK-Taste wird betätigt]
02-21 10:56:03.344: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onPause()
02-21 10:56:03.504: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onStop()
02-21 10:56:03.504: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onDestroy()
```

Wir sehen das dadurch, dass unsere Activity noch »da« war, die Activity **nicht** neu erstellt wird, sondern wieder gestartet (`onRestart()`) wird.

4. Starten über den Launcher, Bildschirm drehen und mit der BACK-Taste beenden

Listing 3.109: **Lebenszyklus Log-File 4**

```
02-21 10:56:31.855: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onCreate()
02-21 10:56:31.884: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onStart()
02-21 10:56:31.895: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onResume()
[Hier wird der Bildschirm gedreht]
02-21 10:56:39.214: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onPause()
02-21 10:56:39.214: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onStop()
02-21 10:56:39.214: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onDestroy()
[Bildschirm wurde gedreht]
02-21 10:56:39.254: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onCreate()
02-21 10:56:39.324: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onStart()
02-21 10:56:39.324: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onResume()
[BACK-Taste]
02-21 10:56:46.094: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onPause()
```

```
02-21 10:56:46.514: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onStop()
02-21 10:56:46.525: DEBUG/de.androidpraxis.marblegame.MarbleGame(1096):
de.androidpraxis.marblegame.MarbleGame.onDestroy()
```

5. Starten und Überlagern mit anderer Activity (z.B. eingehender Anruf oder Benachrichtigung)

Listing 3.110: **Lebenszyklus Log-File 5**

```
02-21 11:09:31.055: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onCreate()
02-21 11:09:31.084: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onStart()
02-21 11:09:31.084: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onResume()
[Hier überlagert eine andere Activity]
02-21 11:09:52.474: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onPause()
02-21 11:09:52.554: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onStop()
[Die andere Activity wird beendet (z.B. BACK-Taste)]
02-21 11:10:26.974: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onRestart()
02-21 11:10:26.974: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onStart()
02-21 11:10:26.985: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onResume()
[BACK-Taste]
02-21 11:10:36.975: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onPause()
02-21 11:10:37.165: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onStop()
02-21 11:10:37.175: DEBUG/de.androidpraxis.marblegame.MarbleGame(1163):
de.androidpraxis.marblegame.MarbleGame.onDestroy()
```

Unsere erste Activity haben wir ja schon über den Projekterstellungsassistenten angelegt. Diese Activity wurde dann auch in das Manifest aufgenommen und mit dem Intent-Filter versehen, um die Activity vom Application Launcher aus starten zu können. Zusätzlich legt der Assistent in den Layouts die Layout-Datei `main.xml` an sowie die Strings für den `app_name` und die unvermeidliche Begrüßung »Hello World«.

Das ist doch schon mal ein guter Ausgangspunkt. An dieser Stelle wollen wir uns noch ein bisschen genauer damit befassen, was man in einer Activity noch so alles anstellen kann.

Weiter oben haben wir uns den Lebenszyklus der Activity betrachtet, jetzt gehen wir daran, die Activity mit Funktionalität zu füllen.

Ich möchte hier weiterhin die Spielwiese benutzen, damit wir relativ schnell etwas sehen und die Dinge ausprobieren können.

Die Spielwiese dient für mich als das, was der Name bereits suggeriert: als Spielwiese. Hier hängen wir einfach nach und nach die Komponenten ein die wir ausprobieren wollen und bauen das Ganze später zur geplanten Anwendung um, die ein Scrapbook realisieren soll.

Wie die `Spielwiese3`-Activity entstanden ist, haben wir bereits am Anfang dieses Kapitels durchgesprochen. In der Zwischenzeit hat sich daran ein bisschen was getan, und zwar ist `Spielwiese3` nicht mehr von `Activity` abgeleitet, sondern von der Klasse `ListMenuActivity`. Die stammt von mir und dient dazu, recht schnell und einfach in einer `ListView` weitere Activities mit einem Namen einzuhängen. `ListMenuActivity` selbst ist wiederum von `ListActivity` abgeleitet, und diese Klasse wird vom Android-System zur Verfügung gestellt. Hinter der `ListActivity` steckt eine Klasse, die bereits ein Layout beinhaltet und weitere Funktionen einführt, um eine ListView darzustellen und auf die Auswahl von Einträgen in der Liste zu reagieren.

```
public class ListMenuActivity extends ListActivity
{
```

Hier deklarieren wir die Klasse ListMenuActivity als Erweiterung von ListActivity:

```
public String EXTRA_SUBMENUE = "de.androidpraxis.ApplicationLibrary.SUBME➥
    NUE";
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);

Intent intent = getIntent();
ListMenue menue = (ListMenue)intent.getSerializableExtra(EXTRA_SUBMENUE);
if (menue == null)
{
    menue = new ListMenue();
    getMainMenue(menue);
}

new SimpleAdapter(this, menue.getList(),
R.layout.mainlist_item, new String[] { "Title" },
new int[] { R.id.mainlist_item_text}));

getListView().setTextFilterEnabled(true);
}
```

In der Methode `onCreate` initialisieren wir die Activity. ListMenue ist eine Hilfsklasse, die es mir erlaubt, einfach Einträge zu definieren und entweder direkt ein Intent, einen Klassennamen oder eine ID hinter den Eintrag zu legen. Das ListMenue kapselt einfach eine Liste, mit der eine ListActivity direkt umgehen kann. Dazu wird mittels `setListAdapter` die Liste an die ListView gebunden.

```
public void getMainMenue(ListMenue menue) {

}
```

Die Methode `getMainMenu(…)` wird später z.B. in der Activity `Spielwiese3` überschrieben und füllt das Objekt `menue` mit entsprechenden Einträgen.

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {
Map map = (Map) l.getItemAtPosition(position);

if (map.containsKey("Command"))
{
  int cmd = (Integer)map.get("Command");
  onMenuCommand(cmd);
}
if (map.containsKey("Intent"))
{
  Intent intent = (Intent) map.get("Intent");
  startActivity(intent);
}
if (map.containsKey("Menue"))
{
ListMenue listMenue = (ListMenue) map.get("Menue");
Intent intent = new Intent(this,this.getClass()).putExtra(EXTRA_SUBMENUE,
listMenue);
startActivity(intent);
}
if (map.containsKey("Class"))
{
String clazz  = (String) map.get("Class");
Intent intent = new Intent();
intent.setClassName(this, clazz);
startActivity(intent);
}
}
```

onListItemClick(…) ist eine Methode, die durch die Klasse ListActivity eingeführt und hier überschrieben wird. Der ListAdapter, der die Daten für die ListView kapselt, liefert als Listeneintrag über l.getItemAtPosition(position) eine Map zurück. Die Map selbst repräsentiert ein assoziatives Array, in dem zu Namen bestimmte Objekte gespeichert werden. Ich speichere in der ListMenue-Klasse in jedem Eintrag einen Wert mit Namen »Titel«, der den Namen des Menüeintrags beinhaltet. Weiterhin steckt in der Map entweder ein Intent, ein Klassenname oder eine ID. Je nachdem was der Menüeintrag beinhaltet, werden unterschiedliche Methoden aufgerufen, um die Aktion hinter dem Menüeintrag auszuführen. Unter anderem sehen wir hier auch die Methode startActivity(…), falls der Menüeintrag ein Intent oder den Klassennamen einer Activity transportiert.

```
public void onMenuCommand(int cmd) {
// TODO Auto-generated method stub

}
```

Falls kein Intent oder Klassenname angegeben ist, wird alternativ die Methode onMenuCommand aufgerufen, die z.B. in Spielwiese3 überschrieben wurde und mittels der ID eine Aktion auslöst.

Listing 3.111: **Eine ListActivity als Basis für ein flexibles Menüsystem**

```
}
```

3.13 Fragments

Fragmente sind eine Neuerung von Android 3. Fragmente stellen einen Teil der Funktionalität und der Benutzeroberfläche innerhalb einer Activity dar. Das Layout für ein Fragment wird in einer eigenen Layout-Ressource definiert. Eine Activity kann nun aus einem oder mehreren Fragmenten bestehen, und die einzelnen Fragmente können zur Laufzeit einfach ausgetauscht werden.

Ähnlich wie der grundsätzliche Entwurf des Fenstersystems in Android ist auch die Einführung der Fragmente in erster Linie den unterschiedlichen unterstützten Bildschirmgrößen geschuldet, die mit der Portierung von Android auf tablettartige Geräte noch um Bildschirmdiagonalen von 7 Zoll bis 10,1 Zoll, und in Zukunft vielleicht noch größere, bereichert wurden.

Diese Bildschirmgrößen bieten wesentlich mehr Platz als die Smartphone-Bildschirme, und das hat Auswirkungen auf den Aufbau der Benutzeroberflächen. Navigieren wir auf einem Smartphone in der Regel über einzelne Fenster, die sich bei Auswahl einer Funktion oder eines Listeneintrags übereinander legen, kann man auf dem Tablett wesentlich effizienter mit dem Platz umgehen und z.B. die Detailansicht zu einem Listeneintrag direkt daneben anzeigen lassen, so wie wir es z.B. von E-Mail-Programmen auf dem PC gewöhnt sind. Man nennt diese Darstellung auch *Dualpane* (für eine zweigeteilte Sicht) oder *Multipane* (für mehrere Bestandteile).

Diese geeilte Ansicht ist ein zentrales Bedienkonzept auf den Tabletts. Neben dem Mail-Client bedienen sich z.B. die Einstellungen und einige andere Anwendungen dieser Technik.

Bisher hätte man dafür mehrere Activities benötigt, eine für die Listendarstellung, eine für die Detaildarstellung und ggf. eine für Tabletts, die beide Darstellungen kombiniert. Mit dem reinen Activity-Konzept bedeutet das, wenn man nicht einen großen Aufwand betreibt und Funktionalitäten in weitere gemeinsame Klassen auslagert, die Verwaltung von in großen Teilen redundantem Quellcode. Mit dem Fragment-Konzept können wir nun ein Fragment für die Listendarstellung schreiben und eines für die Detaildarstellung. Die Fragmente erhalten die entsprechende Funktionalität. Nun können wir diese beiden Fragmente entweder in jeweils zwei Activities (kleiner Bildschirm) oder zusammen in einer Activity (großer Bildschirm) verwenden.

Neben der Bildschirmgröße können wir dann natürlich auch die Ausrichtung des Bildschirms mit einbeziehen. Die Darstellung als Dualpane macht hochkant möglicherweise keinen Sinn, sondern ist dem Querformat vorbehalten. Auch in diesem Fall können wir die zwei Fragmente jeweils unterschiedlich einsetzen (eine oder zwei Activities), und die Auswahl des jeweiligen Layouts regeln wir über die konfigurationsspezifischen Ressourcen (`res\layout-land`, `res\layout-port`)

Der Lebenszyklus der Fragmente innerhalb einer Activity ist an den Lebenszyklus der Activity gekoppelt. Es finden sich in jedem Fragment auch entsprechende Methoden, die während des Lebenszyklus aufgerufen werden.

Ein Fragment ist einer Activity ziemlich ähnlich, sodass die Umstellung auf das Fragment-Konzept nicht sonderlich schwierig ist, aber durch die Modularisierung immense Vorteile bringt.

Neben der Möglichkeit der geteilten Ansicht können Fragmente auch sehr einfach mit Tab-Reitern oder einer Breadcrumb-Navigation in der Action Bar versehen werden. Dadurch lässt sich die Navigation zwischen Fragmenten sehr schön an zentraler Stelle unterbringen.

TIPP

Wir sollten in Zukunft immer Fragmente für die einzelnen zusammengehörenden Bestandteile der Anwendung nutzen, um die Fragmente in Einzelansichten oder zusammengesetzten Ansichten, je nach Gerät und Ausrichtung, nutzen zu können.

Sinnvoll ist dieses Layout immer dann, wenn ich zu einer Auswahlliste (Kontakteinträge, Liste von Bilder, Übersicht über E-Mails) eine Detailansicht habe (Details zum Kontakt, die große Vorschau des ausgewählten Bildes, den Text der E-Mail).

Glücklicherweise können wir Fragmente auch mit Android-Versionen vor Android 3 nutzen, da Google eine Kompatibilitätsbibliothek bereitstellt, die einige Funktionen von Android 3 auch für vorherige Android-Versionen zur Verfügung stellt.

3.13.1 Die Kompatibilitätsbibliothek

Die Bibliothek für die Abwärtskompatibilität befindet sich im Pfad `C:\Programme\Android\android-sdk-windows\extras\android\compatibility\v4` (bzw. dem jeweiligen Installationspfad des SDK Managers). Falls die Bibliothek noch nicht vorhanden ist, laden wir sie mit dem SDK-Manager herunter. Sie hört auf den Namen *Android Compatibility Package* und befindet sich im Android Repository.

Die JAR-Datei `android-support-v4.jar` kopieren wir in ein Verzeichnis `libs`, das wir unterhalb von unserem Projekt angelegt haben. In der Eclipse markieren wir die Bibliothek und führen den Menüpunkt *Build Path* → *Add to Buildpath* des Kontextmenüs aus. Damit wird die Bibliothek in unsere Projektkonfiguration aufgenommen. Zu beachten ist hierbei, dass, wenn wir die Bibliothek in einem Android-Library-Projekt benutzen, wir die Bibliothek auch zum eigentlichen Projekt, das unsere Android-Library nutzt, hinzufügen.

Mit hilfe dieser Bibliothek können wir nun auch für die Android-Versionen ab 1.5 Fragmente nutzen. Dabei gelten allerdings ein paar Regeln:

1. Wir müssen eine FragmentActivity benutzen, um Fragmente einzubinden (das ist unter 3 nicht nötig, das kann die Activity von sich aus).
2. Innerhalb der Activity greifen wir über FragmentActivity.getSupportFragmentManager() auf den Fragment-Manager zu, unter Android 3 reicht Activity.getFragmentManager().
3. Die ActionBar wird nicht unterstützt, es ist aber möglich, über die Hilfsklasse Menu-Compat Menüeinträge entsprechend im Programm zu konfigurieren.

INFO

Da es wohl noch eine Zeit dauern wird, bis die Entwicklungslinien wieder verschmelzen und auch noch eine gewisse Zeit Geräte mit älterer Android-Version am Markt sein werden, ist der Einsatz der Kompatibilitätsbibliothek sicher empfehlenswert. Adressieren wir hingegen tatsächlich nur Geräte ab Version 3, können wir auf die Kompatibilitätsbibliothek verzichten.

Die Spielwiese ist so organisiert, dass möglichst viele gemeinsame Funktionalitäten in der SpielwieseLibrary zusammengefasst werden, die als Zielplattform den Level 8 (Android 2.2) definiert. Um hier bereits Fragmente nutzen zu können, bindet die SpielwieseLibrary die Kompatibilitätsbibliothek mit ein. Spezielle Funktionalitäten, die z.B. nur auf Android 3 laufen, werden dann jeweils in den Projekten für die spezifische Zielplattform gepackt.

3.13.2 Fragmente im Detail

Schauen wir uns das Fragment im Detail an. Die folgende Übersicht zeigt die wichtigsten Methoden, die wir für unsere Fragmente benutzen.

onAttach(Activity)	Wird aufgerufen, wenn das Fragment in die Activity eingehängt wird. Wird vor onCreate(...) aufgerufen.
onCreate(Bundle)	Wird aufgerufen, wenn das Fragment initialisiert werden soll. Aufruf erfolgt nach onAttach(...) und vor onCreateView(...). **Achtung:** Hier liegt der Unterschied zu den Activities. In der Activity setzen wir das Layout innerhalb von onCreate(...). Das Fragment liefert sein Layout in onCreateView(...) zurück.
onCreateView(LayoutInflater, ViewGroup, Bundle)	Wird aufgerufen, damit das Fragment sein Layout erstellt und zurückliefert. Ein LayoutInflater und die ViewGroup, in die das Layout eingehängt wird, werden übergeben. Kann auch null zurückliefern, wenn das Fragment kein Userinterface hat.

Tabelle 3.41: **Die wichtigsten Callbacks eines Fragments**

onActivityCreated(Bundle)	Wird unmittelbar nach dem onCreate(...) der Activity aufgerufen. Die Activity ist zu diesem Zeitpunkt komplett initialisiert.
onStart()	Das Fragment wird sichtbar (in Folge von onStart() der Activity).
onResume()	Das Fragment kann mit dem Benutzer interagieren, also Eingabeereignisse entgegennehmen.
onPause()	Das Fragment nimmt keine Eingabeereignisse mehr an. **Wichtig:** Analog zur Activity ist das der Zeitpunkt, in dem zum einen ressourcenintensive Aktivitäten gestoppt oder austehende Änderungen (in Datenbankeinträgen oder Dateien) gespeichert werden sollten.
onStop()	Das Fragment ist nicht mehr sichtbar.
onDestroyView()	Wird aufgerufen, bevor die View des Fragments zerstört wird. Hier kann man Ressourcen aufräumen, die mit der View verbunden sind, z.B. Bitmaps recyclen.
onDestroy()	Das Fragment steht kurz vor seiner endgültigen Zerstörung.
onDetach()	Das Fragment wird von der Activity entfernt (z.B. durch eine Fragmenttransaktion, bei der ein neues Fragment statt diesen Fragments eingehängt wird).
onConfigurationChanged (Configuration newConfig)	Wird analog zur Activity aufgerufen, wenn die Gerätekonfiguration verändert wird (Ausrichtung wird gewechselt etc.).
onCreateContextMenu (ContextMenu menu, View v, ContextMenu.ContextMenuInfo menuInfo)	Wird aufgerufen, wenn eine View ein Kontextmenü anfordert. Diese Methode wird immer vor dem Anzeigen des Kontextmenüs ausgeführt. Die View muss dazu mittels registerForContextMenu (View) registriert sein.
onCreateOptionsMenu(Menu menu, MenuInflater inflater)	Wird aufgerufen, um das Optionenmenü zu erzeugen. Das Fragment muss per setHasOptionsMenu(boolean) anzeigen dass es Menüeinträge erstellen will.
onOptionsItemSelected (MenuItem item)	Wird aufgerufen, wenn ein Menüeintrag im Optionenmenü ausgewählt wurde.

Tabelle 3.41: **Die wichtigsten Callbacks eines Fragments (Forts.)**

onPrepareOptionsMenu (Menu menu)	Wird aufgerufen, bevor das Optionenmenü angezeigt wird. Hier kann man z.B. Menüeinträge deaktivieren o.Ä.
onContextItemSelected (MenuItem item)	Wird aufgerufen, wenn ein Menüeintrag aus dem Kontextmenü aufgerufen wird. ListFragment führt eine weitere Methode ein, die den ausgewählten Listeneintrag übergibt.

Tabelle 3.41: **Die wichtigsten Callbacks eines Fragments (Forts.)**

Betrachten wir das Fragment anhand des `ImageViewFragment`.

```
public class ShowImageFragment extends Fragment {
private Bitmap bitmap = null;
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setHasOptionsMenu(true);
}
```

Initialisieren des Fragments und bekanntmachen, dass unser Fragment ein Optionenmenü hat.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
    View result = inflater.inflate(R.layout.showimagelayout, container, fal➥
       se);
    ImageView imageView = (ImageView)result.findViewById(R.id.showimage_
imageview);
    if (imageView!=null)
    {
        registerForContextMenu(imageView);
    }
    if (getArguments()!=null && getArguments().containsKey("data"))
    {
    if (imageView!=null)
    {
    imageView.setImageURI(Uri.parse(getArguments().getString("data")));
    }
    }
    return result;
}
```

Erstellen der View mittels des LayoutInflaters. Es wird noch geprüft, ob externe Argumente beim Erstellen des Fragments mitgegeben wurden (getArguments()), indem der URI des Bildes übergeben wurde, das angezeigt werden soll. Das ist nützlich, wenn wir für jede Bildanzeige einfach ein neues Fragment erstellen und das alte Fragment ersetzen. Die View wird für das Erstellen eines Kontextmenüs registriert.

```
public void setImageURI(Uri uri) {
    ImageView imageView = (ImageView)this.getView().findViewById(R.
id.showimage_imageview);
    if (imageView!=null)
    {
        InputStream is;
        try {
        if (bitmap!=null)
        {
        bitmap.recycle();
        }
        is = getActivity().getApplication().getContentResolver().
openInputStream(uri);
        bitmap = BitmapFactory.decodeStream(is);
        is.close();
        imageView.setImageBitmap(bitmap);
        } catch (FileNotFoundException e) {
        } catch (IOException e) {
        }
    }
}
```

Setzen des URI des Bildes, das angezeigt werden soll. Wichtig ist das `bitmap.recycle()`, um die vorherige Bitmap genau jetzt aus dem Speicher zu werfen. Bemerkenswert ist hier, dass wir die Auflösung des Content-URI an den Content Resolver übergeben. Dadurch wird es vollkommen transparent, woher unser Bild stammt, der Content Resolver sucht den entsprechenden Provider, der uns einen Stream auf die Bilddaten zurückliefert.

```
static ShowImageFragment createImageFragment(Uri uri)
{
    ShowImageFragment f = new ShowImageFragment();
    Bundle args = new Bundle();
    args.putString("data", uri.toString());
    f.setArguments(args);
    return f;
}
```

Diese statische Methode dient zum Erzeugen eines neuen Fragments mit dem entsprechend assoziierten URI des Bildes. Hier sieht man auch, wie die Argumente besetzt werden, die nachher in onCreateView() abgefragt werden können. Diese statische Methode wird benutzt, wenn wir jedes Bild durch ein neues Fragment darstellen wollen.

```
@Override
public void onDestroyView() {
    super.onDestroyView();
    if (bitmap!=null)
    {
        bitmap.recycle();
    }
}
```

Wenn die View des Fragments zerstört wird (z.B. durch einen Neustart nach Lageänderung), müssen wir die verwendete Bitmap recyceln, um eine unnötige Speicherbelastung zu vermeiden.

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    Toast.makeText(getActivity(), "Ausgewählt: "+item.getTitle(), Toast.
LENGTH_LONG).show();
    return super.onContextItemSelected(item);
}
```

Reaktion auf die Auswahl eines Eintrags aus dem Kontextmenü.

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    MenuInflater inflater = getActivity().getMenuInflater();
    inflater.inflate(R.menu.imageviewfragment_context_menu, menu);
}
```

Erstellen des Kontextmenüs.

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {

    super.onCreateOptionsMenu(menu, inflater);
    inflater.inflate(R.menu.imageviewfragment_context_menu, menu);
    MenuCompat.setShowAsAction(menu.findItem(R.id.item_move_to_folder), 1);
}
```

Erstellen des Optionenmenüs. Über MenuCompat können Menüeinträge in der ActionBar aktiviert werden.

Listing 3.112: **Fragment, um ein Bild anzuzeigen**

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    Toast.makeText(getActivity(), "Ausgewählt: "+item.getTitle(), Toast.
LENGTH_LONG).show();
    return super.onOptionsItemSelected(item);
}
```

Reaktion auf die Auswahl eines Eintrags aus dem Optionenmenü.

```
}
```

Und so sieht das Layout zum Fragment aus. Wie wir sehen, werden die Layouts für Fragmente genauso definiert wie für Activities.

Listing 3.113: **Layout zum Fragment**

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_height="match_parent" android:layout_width="match_parent">
    <ImageView android:src="@drawable/icon"
    android:hapticFeedbackEnabled="true"
    android:id="@+id/showimage_imageview"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"></ImageView>
</FrameLayout>
```

Wie bekommen wir nun das Fragment in die Activity? Grundsätzlich gibt es dafür zwei Möglichkeiten:

1. Per Layout
2. Über die Programmierung

In der Regel wird man die Fragmente über das Layout einbinden. Da wir die Layouts konfigurationsspezifisch ablegen können, erreichen wir das Ziel, eine flexible Oberfläche zu bauen, damit sehr schnell.

Listing 3.114: **Die Activity, die die Fragmente einbettet**

```
public class ShowImagesWithFragmentsActivity extends FragmentActivity  {
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.showimageswithfragmentslayout);
}
}
```

INFO

Hier ist die Activity wegen der Verwendung der Kompatibilitätsbibliothek von `FragmentActivity` **abgeleitet. Wenn wir auf die Kompatibilität verzichten, benutzen wir einfach** `public class ShowImagesWithFragmentsActivity extends FragmentActivity`.

Wie wir sehen ist die Activity ziemlich unspektakulär. Es wird lediglich eine Content View gesetzt, die folgendermaßen aussieht:

Listing 3.115: **Layout für kleine Bildschirme bzw. das Standardlayout (res\layout)**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment class="de.androidpraxis.SpielwieseLibrary.ShowImagesFragment"
    android:id="@+id/showimagesfragment"
    android:layout_width="match_parent" android:layout_height="match_parent"
/>
</LinearLayout>
```

Listing 3.116: **Layout für Bildschirme im Querformat (res\layout-land)**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment class="de.androidpraxis.SpielwieseLibrary.ShowImagesFragment"
    android:id="@+id/showimagesfragment"
    android:layout_weight="1"
    android:layout_width="0dp" android:layout_height="match_parent" />
    <fragment class="de.androidpraxis.SpielwieseLibrary.ShowImageFragment"
    android:id="@+id/showimagefragment"
    android:layout_weight="3"
    android:layout_width="0dp" android:layout_height="match_parent" />
</LinearLayout>
```

Listing 3.117: **Layout für große Bildschirme im Hochformat (res\layout-xlarge-port)**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment class="de.androidpraxis.SpielwieseLibrary.ShowImageFragment"
    android:id="@+id/showimagefragment"
    android:layout_weight="1"
    android:layout_height="0dp" android:layout_width="match_parent" />
    <fragment class="de.androidpraxis.SpielwieseLibrary.ShowImagesFragment"
    android:id="@+id/showimagesfragment"
    android:layout_weight="3"
    android:layout_height="0dp" android:layout_width="match_parent" />
</LinearLayout>
```

Lediglich durch die drei unterschiedlichen Layouts wird das Aussehen der Activity bestimmt. Auf großen Bildschirmen wird sowohl im Querformat als auch im Hochformat das Fragment zur Bildanzeige eingebettet, im Standardlayout nicht. Damit wird im Hochformat auf allen Bildschirmen, außer den wirklich großen Bildschirmen, nur die Liste der Bilder angezeigt. Im Querformat, oder bei großen Bildschirmen auch im Hochformat, wird auf allen Bildschirmen zusätzlich die Vorschau angezeigt (was selbstverständlich nicht optimal ist, wenn der Bildschirm unter 4 Zoll Bilddiagonale misst).

ACHTUNG

Die obigen Beispiele hängen wir die Fragmente direkt in das Layout ein. Wenn wir später mittels des `FragmentManager` und der `FragmentTransaction` z.B. die Detailansicht austauschen wollen und nicht, wie in diesem Beispiel, das eingehängte Fragment immer wieder verwenden, empfiehlt es sich, die Fragmente nicht direkt im Layout einzuhängen, sondern einen Container bereitzustellen und die Fragmente innerhalb der Anwendung mittels FragmentManager hinzuzufügen. Die Manipulation bereits im Layout eingehängter

ACHTUNG

Fragmente ist nicht konsistent und führt zu merkwürdigen Effekten. Vielleicht ist das ein Bug oder ein Feature, auf jeden Fall ist es leichter zu handhaben, wenn wir in diesem Fall tatsächlich die Fragmente programmtechnisch erstellen und einhängen.

Die Fragmente werden nun einfach durch die `<fragment class="…"/>`-Elemente in den Layouts eingebunden und können später durch den Fragment-Manager über ihre ID angesprochen werden, z.B. um die Detailansicht zu aktualisieren:

Listing 3.118: **Auszug aus dem Fragment ShowImagesFragment**

```
void showDetails(int position)
{
    if (position<getListView().getCount())
    {
        currentSelection = position;
        long id = getListView().getItemIdAtPosition(position);
        Uri uri = ContentUris.withAppendedId(contentUri, id);
        if (hasDualPane)
        {
        getListView().setSelection(position);
        getListView().setItemChecked(position, true);

        Fragment detailFragment = getFragmentManager().findFragmentById
(R.id.showimagefragment);
        if (detailFragment!=null)
        {
        ((ShowImageFragment)detailFragment).setImageURI(uri);
        }
        }
        else
        {
        startActivity(new Intent(getActivity(),ShowImageActivity.class).
putExtra("data",uri.toString()));
        }
    }
}
```

Im obigen Listing aktualisieren wir die Detailansicht, falls eine zweigeteilte Ansicht (`hasDualPane`) existiert, ansonsten wird mittels Intent die Activity für die Detailanzeige gestartet, die sich dann über die Liste legt.

Ob eine zweigeteilte Ansicht existiert, ermitteln wir z.B. darüber, ob die View mit der ID *`showimagefragment`* vorhanden ist:

Listing 3.119: **Ermitteln, ob eine zweigeteilte Ansicht vohanden ist**

```
View detailView = getActivity().findViewById(R.id.showimagefragment);
hasDualPane = detailView!=null && detailView.getVisibility() == View.VISIB➥
    LE;
```

3.13.3 FragmentManager und FragmentTransaction

Betrachten wir uns noch die Klassen `FragmentManager` und `FragmentTransaction`. Der `FragmentManager` bietet den Zugriff auf die Fragmente und erlaubt es, sogenannte Transaktionen zu starten. Die Transaktionen dienen dazu, neue Fragmente in die Hierarchie einzuhängen, Fragmente auszutauschen (bzw. übereinander zu legen), Fragmente zu verstecken oder wieder sichtbar zu machen und Fragmente zu entfernen. Darüber hinaus können auch noch Animationen auf die Fragment-Transaktion angewendet werden, um z.B. den Wechsel zwischen den Detailansichten noch etwas spannender zu gestalten.

Über den `FragmentManager` und die `FragmentTransaction` ist es darüber hinaus möglich, Fragmente programmtechnisch in ein Layout einzuhängen, ohne das Fragment im Layout-File zu definieren. Wir benötigen lediglich einen Container, in den das oder die Fragmente innerhalb unserer Anwendungsoberfläche eingehängt werden sollen. Außerdem können wir auch Fragmente **ohne** Benutzeroberfläche einhängen. Das ist immer dann nützlich, wenn wir ein Fragment als »Hintergrundarbeiter« einsetzen wollen. Das könnte z.B. ein Fragment sein, das im Hintergrund die Vorschauansichten für geladene Bilder generiert und dazu das *Loader*-Konzept benutzt. Loader sind (ab Version 3) auf Activities und Fragmenten verfügbar. Mit der Kompatibilitätsbibliothek ist das Konzept auch auf ältere Versionen übertragbar und entbindet uns von der ständigen Neuerfindung von Hintergrund-Threads zum geschmeidigen Laden großer Datenmengen.

Zugriff auf den `FragmentManager` erhalten wir innerhalb einer Activity oder eines Fragments mittels `Activity.getFragmentManager()` bzw. `Activity.get`Support`FragmentManager()` sowie `FragmentManager.getFragmentManager()`.

Der so erhaltene `FragmentManager` ist mit unserem Kontext verknüpft, also mit der Activity, und verwaltet alle Fragmente innerhalb dieser Activity, aber nicht über die Grenze der Activity hinweg.

void addOnBackStack-ChangedListener (FragmentManager. OnBackStackChanged-Listener listener)	Registrieren eines Listeners, der auf Änderungen des Back-Stacks horcht.
FragmentTransaction beginTransaction()	Startet eine Transaktion
boolean executePending-Transactions()	Ausführen ausstehender Transaktionen nach Beenden einer Transaktion. Das ist in der Regel nicht nötig, die Transaktionen werden im Thread des User-Interface ausgeführt sobald die Abarbeitung dort landet. **Tipp:** Dieser Aufruf ist dann sinnvoll, wenn durch eine Transaktion ein Fragment entfernt und ein anderes auf den Stapel gelegt wird. Wird das nämlich gemacht, solange die aktuelle Transaktion noch nicht ausgeführt wurde, kann es zu einem ungültigen Zustand kommen.

Tabelle 3.42: **Methoden des FragmentManagers**

Fragment findFragment-ById(int id)	Sucht das Fragment mit der gegebenen ID.
Fragment findFragment-ByTag(String tag)	Sucht das Fragment mit dem gegebenen Tag. Wird meistens für Fragmente benutzt, die nicht im Layout eingehängt werden und selbst kein Layout – und damit auch keine ID – haben.
FragmentManager. BackStackEntry getBack-StackEntryAt (int index)	Liefert den Inhalt des Backstacks für den gegebenen Index zurück. BackStackEntry liefert dann die ID des Fragments sowie den Breadcrumb-Pfad (Brotkrümel-pfad, dazu später mehr).
int getBackStackEntry-Count()	Liefert die Anzahl der Einträge im Backstack.
Fragment getFragment(Bundle bundle, String key)	Liefert eine Fragment-Instanz, die im Bundle (mittels putFragment(...)) gespeichert wurde. Damit kann man Fragmente mit Bundles transportieren, z.B. als Argument zu anderen Fragmenten.
void popBackStack()	Entfernt den obersten Eintrag aus dem Backstack. Wirkt wie das Drücken der BACK-Taste.
abstract void popBackStack(String name, int flags)	Entfernt die Einträge aus dem Backstack bis zu der Transition mit dem Namen name (der Name wird bei addToBackStack(...) in der Transaktion festgelegt). Wird als Flag FragmentManager. POP_BACK_STACK_INCLUSIVE angegeben, wird auch diese Transition entfernt, ansonsten nicht.
abstract void popBackStack(int id, int flags)	Entfernt die Einträge aus dem Backstack bis zu der Transition mit der ID id (die ID wird bei commit(...) in der Transaktion zurückgeliefert). Wird als Flag Fragment-Manager. POP_BACK_STACK_INCLUSIVE angegeben, wird auch diese Transition entfernt, ansonsten nicht.
abstract boolean pop-BackStackImmediate (int id, int flags)	Wie die obigen Methoden, diese werden allerdings sofort ausgeführt, die obigen erst dann, wenn die Hauptereignisschleife der Anwendung wieder betreten wird.
abstract boolean pop-BackStackImmediate (String name, int flags)	
abstract boolean pop-BackStackImmediate()	
abstract void putFragment(Bundle bundle, String key, Fragment fragment)	Speichert das Fragment in einem Bundle, z.B. um das Fragment als Argument an ein anderes zu übergeben.

Tabelle 3.42: **Methoden des FragmentManagers (Forts.)**

abstract void remove-OnBackStackChanged-Listener (FragmentManager. OnBackStackChanged-Listener listener)	Entfernt den Listener.

Tabelle 3.42: **Methoden des FragmentManagers (Forts.)**

Die wichtigsten Methoden sind:

1. beginTransaction()
2. findFagmentById(...)
3. findFragmentByTag(...)

Der Aufruf von beginTransaction() liefert eine FragmentTransaction zurück, innerhalb der die Fragmente manipuliert werden können.

FragmentTransaction	add(int containerViewId, Fragment fragment)	Hängt ein Fragment in den Container mit der ID containerViewId ein.
FragmentTransaction	add(Fragment fragment, String tag)	Fügt ein Fragment mit dem Tag tag zum FragmentManager hinzu. Das Fragment wird nicht in das Layout eingehängt und erstellt selbst auch kein Layout (on-CreateView wird nicht aufgerufen).
FragmentTransaction	add(int containerViewId, Fragment frag-ment, String tag)	Hängt ein Fragment in den Container mit der ID containerViewId ein und gibt dem Fragment ein Tag mit.
FragmentTransaction	addToBackStack (String name)	Fügt die Transatkion zum Backstack hinzu. Das heißt, der aktuelle Zustand wird auf den Stack gelegt und kann dann per BACK-Navigation, über die Bread-crumb oder per Methodenaufruf wieder zurückgefahren werden. Optional kann ein Name für den Zustandsübergang vergeben werden.
int Schedules a commit of this transaction.	commit()	Schließt die Transaktion ab. Die Operati-onen innerhalb der Transaktion werden ausgeführt sobald die Kontrolle wieder an den UI-Thread geht oder execute-PendingTransactions() auf dem Frag-mentManager aufgerufen wird.

Tabelle 3.43: **Methoden von FragmentTransaction**

int	commit-Allowing-StateLoss()	Wie commit(), allerdings kann dies auch nach dem Speichern des Zustands der Activity erfolgen, commit() selbst darf das nicht.
FragmentTransaction	disallowAddTo-BackStack()	Verhindert die Ablage der Transaktion auf dem Backstack. Sollte innerhalb der Transaktion doch addToBackStack() aufgerufen werden, wird ein Ausnahmefehler erzeugt.
FragmentTransaction	hide(Fragment fragment)	Das Fragment wird versteckt.
boolean	isAddToBack-StackAllowed()	Ermittelt, ob der Zustand auf dem Backstack abgelegt werden darf.
boolean	isEmpty()	Ist die Transaktion leer?
FragmentTransaction	remove (Fragment fragment)	Entfernt das Fragment
FragmentTransaction	replace(int containerViewId, Fragment fragment, String tag)	Ersetzt das Fragment im Container mit der gegebenen Container-ID durch das neue Fragment und gibt dem neuen Fragment einen Tag.
FragmentTransaction	replace(int containerViewId, Fragment fragment)	Ersetzt das Fragment im Container mit der gegebenen Container-ID durch das neue Fragment.
FragmentTransaction	setBreadCrumb-ShortTitle(int res)	Setzt den Kurztitel der Breadcrumb-Navigation. Sinnvoll in Verbindung mit FragmentBreadCrumbs.
FragmentTransaction	setBreadCrumb ShortTitle(Char-Sequence text)	
FragmentTransaction	setBreadCrumb-Title(Char-Sequence text)	Setzt den Titel der Breadcrumb-Navigation. Sinnvoll in Verbindung mit FragmentBreadCrumbs.
FragmentTransaction	setBreadCrumb-Title(int res)	
FragmentTransaction	setCustom-Animations(int enter, int exit)	Setzen einer Animation die auf die Fragmente in der Transaktion angewendet werden soll.

Tabelle 3.43: **Methoden von FragmentTransaction (Forts.)**

FragmentTransaction	setTransition(int transit)	Setzen einer Standardanimation für den Zustandsübergang. FragmentTransaction.TRANSIT_FRAGMENT_CLOSE Animation für Schließen eines Fragments (remove) FragmentTransaction.TRANSIT_FRAGMENT_FADE Animation für Anzeigen oder Verstecken eines Fragments (hide/show) FragmentTransaction.TRANSIT_FRAGMENT_OPEN Animation für Öffnen eines Fragments (add) FragmentTransaction.TRANSIT_NONE Keine Animation
FragmentTransaction	setTransition-Style(int style-Res)	Setzt eine Stilressource die bei der Animation des Übergangs angewendet wird.
FragmentTransaction	show(Fragment fragment)	Anzeigen des vorher unsichtbaren Fragments.

Tabelle 3.43: **Methoden von FragmentTransaction (Forts.)**

Listing 3.120: **Zufügen eines Fragments**

```
private void addFragment(boolean addToBackStack)
{
    FragmentManager fragmentManager = getFragmentManager();
    FragmentTransaction ta = fragmentManager.beginTransaction();
    ta.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN);

    Fragment fragment;
    fragment = fragmentManager.findFragmentById(R.id.lifecyclefragment);
    if (fragment!=null)
    {
        ta.remove(fragment);
    }

    fragment = LifecycleFragment.createFragment(instanceNo);
    ta.add(R.id.lifecyclefragment, fragment);

    ta.setBreadCrumbShortTitle("#"+Integer.toString(instanceNo));
    ta.setBreadCrumbTitle("#"+Integer.toString(instanceNo));
    if (addToBackStack) ta.addToBackStack(null);
    ta.commit();
    fragmentManager.executePendingTransactions();
    instanceNo+=1;
}
```

Im obigen Listing wird ein Fragment in den Container mit der ID `R.id.lifecyclefragment` eingehängt, ein vorheriges Fragment wird ggf. entfernt. Im Prinzip würde auch ein `ta.replace(R.id.lifecyclefragment, fragment)` reichen, ich hatte allerdings ein paar merkwürdige Effekte wenn ich nur diesen Aufruf benutzte.

Der Aufruf von `executePendingTransactions()` stellt hier sicher, dass die Transaktion beendet wird, bevor der Button wieder betätigt werden kann. Ohne den Aufruf könnte es passieren, dass eine neue Transaktion gestartet wird, ohne dass die alte bereits abgearbeitet ist. Daraus resultiert dann möglicherweise ein ungültiger Zustand, der zum Fehler führt.

Listing 3.121: **Layout mit dem Fragment – Container**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="match_parent"
android:layout_height="match_parent">
    <FrameLayout
    android:id="@+id/lifecyclefragment"
    android:layout_width="match_parent"
    android:layout_height="0px"
    android:layout_weight="1">
    </FrameLayout>

    <Button android:id="@+id/addfragment" android:layout_height="wrap_con➥
    tent" android:text="@string/addfragment" android:layout_width="match_pa➥
    rent"/>
</LinearLayout>
```

Das Layout stellt mittles `<FrameLayout … />` den Container bereit, in den wir unsere Fragmente einhängen können. Bemerkenswert ist, dass wir hier kein `<fragment … />`-Tag benutzen. Das ist bewusst so realisiert, da wir die Fragmente immer programmtechnisch hinzufügen wollen. Wenn wir ein Fragment benötigen, das bereits beim Start der Activity angezeigt wird, dann fügen wir das einfach in der `onCreate(…)` Methode der Activity hinzu.

3.13.4 Breadcrumbs

Ein Aspekt der Fragmente ist die Kopplung an die Action Bar. Durch die Action Bar bieten die Tabletts einen Bereich, der für allgemeine Menüeinträge genutzt werden kann und darüber hinaus auch Platz für weitere Navigationsmöglichkeiten in den Anwendungen bietet. Das kann z.B. eine Adress- oder Suchzeile sein, aber auch Tabs für mehrseitige Anwendungen oder eine Breadcrumb-Navigation.

Breadcrumbs sind Brotkrümel, und wie im Märchen von Hänsel und Gretel sollen die Brotkrümel den Weg zurück zum Ausgangspunkt zeigen. Glücklicherweise kommen in unseren Anwendungen keine Vögel vorbei, die die Krümel hinter uns aufpicken. Obwohl… ärgerliche Vögel gibt's ja auch auf Android.

Breadcrumb-Navigation wird gerne dort eingesetzt, wo man sich von einer Option zur nächsten hangelt, oder aber auch, wenn man sich in einem Baum von Knoten zu Knoten hangelt. Dabei bewegt man sich ja entlang eines Pfades, der im Prinzip linear ist. Um nun zurück zu gelangen, gehen wir den Pfad einfach rückwärts. Manchmal bewegen wir uns aber auch nicht durch einen Baum (und haben immer nur einen Vorgängerknoten), sondern durch ein Netz, wie zum Beispiel in Form von Hypertexten, an dem in jedem Knotenpunkt beliebig viele Ein- und Ausgänge münden. Hier setzt der Brotkrümelpfad an, der uns immer den Weg zum vorherigen Knoten weist.

Die FragmentTransaction bietet nun die Möglichkeit, den Übergang von einem Platz zum nächsten auf den sogenannten Backstack zu legen. Jede Transaktion kann mit einem Titel gekennzeichnet werden, der dann wiederum jeden Brotkrümel benennt. Mit der BACK-Taste, oder auch programmtechnisch, können wir uns dann wieder im Backstack zurückhangeln, in dem wir einfach den obersten Eintrag runterschmeißen und zum vorherigen Eintrag zurückkehren und so weiter.

Stellen wir nun die Brotkrümel auch dar, können wir aber auch einen beliebigen Vorgänger auswählen, zu dem wir zurückkehren wollen. Es werden einfach alle Einträge des Backstacks abgebaut, bis wir am Zielkrümel angelangt sind.

ACHTUNG

Das bedeutet aber, dass wir nicht mehr beliebig nach vorne springen können. Haben wir uns erst einmal zurückgearbeitet, entweder Schritt für Schritt oder per Breadcrumb-Navigation über mehrere Einträge zurück, können wir nur noch Schritt für Schritt nach vorne gehen.

Um die Brotkrümel darzustellen, führt Android 3 die `FragmentBreadCrumbs`-View ein, die vorzugsweise in der ActionBar platziert wird:

Listing 3.122: **Erstellen einer Breadcrumb-Navigation in der Action Bar**

```
public class LifecycleActivity extends Activity implements OnClickListener
{
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView( R.layout.lifecycleactivity );
    […]
    FragmentBreadCrumbs breadCrumbs = new FragmentBreadCrumbs(this);
    getActionBar().setDisplayOptions(ActionBar.DISPLAY_SHOW_
CUSTOM,ActionBar.DISPLAY_SHOW_CUSTOM);
    getActionBar().setCustomView(breadCrumbs);
    breadCrumbs.setActivity(this);
    breadCrumbs.setTitle("#1", "#1");

    addFragment(false);
}
```

Wichtig ist hier zweierlei. Zum einen muss die `ActionBar` mittels `setDisplayOptions(…)` dafür vorbereitet werden, eine View darzustellen. Zum anderen muss die `BreadCrumb`-View mit unserer Activity verknüpft werden, denn über diesen Kontext erhält die `BreadCrumb`-View Zugriff auf den `FragmentManager` und kann auf die Änderungen des Backstacks reagieren sowie die Navigation durch den Backstack realisieren.

Mit `breadCrumbs.setTitle(<Titel>, <Kurztitel>);` wird die Wurzel der Navigation initialisiert. Damit wird der erste Eintrag betitelt, wenn noch keine Fragmente zugefügt wurden.

Einträge in die Breadcrumb-Navigation finden nur statt, wenn eine Transaktion auf den Backstack gelegt wird und entsprechend einen Titel und/oder Kurztitel erhält:

Listing 3.123: **BreadCrumb-Eintrag erzeugen**

```
ta.setBreadCrumbShortTitle(<Kurztitel>);
ta.setBreadCrumbTitle(<Titel>);
ta.addToBackStack(null);
```

Für jeden Eintrag legt die View einen entsprechenden Navigationspunkt an, der auch antippbar ist und zum jeweiligen Punkt zurückführt.

3.13.5 Tabs

Die Action Bar ist auch ein guter Platz für Tabs, wenn wir in unserer Anwendung eine Ansicht mit mehreren Seiten haben. Das kann z.B. unser persönlicher Media-Player sein, der mit den Tabs die Optionen »Alle«, »Nach Genre«, »Nach Album«, »Nach Interpret« und »Favoriten« anbietet, oder ein Mailclient, der die Optionen »Alle«, »Ungelesen«, »Nach Konversation« als Tabs anbietet.

Um Tabs bereitzustellen, muss die Action Bar entsprechend mit `actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS)` konfiguriert werden. Herzstück der Tab-Verwaltung ist der `ActionBar.TabListener`, den wir nach unseren Bedürfnissen implementieren müssen. Mittels `actionBar.addTab(actionBar.newTab().setText(<TEXT>).setTabListener(<LISTENER>)` fügen wir nach Bedarf Tabs zur Action Bar hinzu. Die Reaktion auf das Antippen eines Tabs implementieren wir im TabListener. Im Listener können wir dann entweder neue Fragmente ins Layout einhängen oder bestehende Fragmente entsprechend der Auswahl manipulieren, es muss also nicht per se für jeden Tab ein eigenes Fragment erstellt werden.

Ein sehr einfaches Beispiel basiert auf unserer `LifecycleActivity`, die wir zu einer `LifecycleTabActivity` umfunktionieren. Hiermit können wir in der Log-Ausgabe auch schön beobachten, wie sich die Lebenszyklen der einzelnen Tabs verhalten.

Listing 3.124: **Erstellen von Tabs in der Action Bar**

```
public class LifecycleTabActivity extends Activity implements OnClickListe➥
    ner{
    private class TabListener implements ActionBar.TabListener {
        private Fragment fragment;
```

```
        public TabListener(Fragment fragment) {
        this.fragment = fragment;
        }

        public void onTabSelected(Tab tab, FragmentTransaction ft) {
        ft.setCustomAnimations(R.anim.fragment_enter_animation2, R.anim.
fragment_exit_animation2);
        ft.add(R.id.tabcontent, fragment, null);
        }

        public void onTabUnselected(Tab tab, FragmentTransaction ft) {
        ft.setCustomAnimations(R.anim.fragment_enter_animation2, R.anim.
fragment_exit_animation2);
        ft.remove(fragment);
        }

        public void onTabReselected(Tab tab, FragmentTransaction ft) {

        }

    }
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
    super.onCreate(savedInstanceState);

    setContentView( R.layout.lifecycletabactivity );

    final ActionBar actionBar = getActionBar();
    actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);

    actionBar.addTab(actionBar.newTab().setText("Tab 1").setTabListener(new
TabListener(LifecycleFragment.createFragment(1))));

    actionBar.addTab(actionBar.newTab().setText("Tab 2").setTabListener(new
TabListener(LifecycleFragment.createFragment(2))));

    actionBar.addTab(actionBar.newTab().setText("Tab 3").setTabListener(new
TabListener(LifecycleFragment.createFragment(3))));

    actionBar.addTab(actionBar.newTab().setText("Tab 4").setTabListener(new
TabListener(LifecycleFragment.createFragment(4))));
}
[…]
}
```

Die Implementierung ist sehr simpel. Mit jedem Tab wird eine Instanz des privaten TabListeners erzeugt, der wiederum eine Instanz eines Fragments erhält. In

```
public void onTabSelected(Tab tab, FragmentTransaction ft) {
    ft.add(R.id.tabcontent, fragment, null);
}
```

wird dann lediglich das Fragment in den Container eingeklinkt und in

```
public void onTabUnselected(Tab tab, FragmentTransaction ft) {
    ft.remove(fragment);
}
```

wieder entfernt.

Eine Speicherplatz schonende Variante wäre, dem Listener einfach einen Index-Wert (1,2,3...) mitzugeben, auf den er reagiert und der das entsprechende Fragment erst bei Bedarf erzeugt.

Weiterhin ist es auch möglich, die `TabListener`-Methoden auf der Activity direkt zu implementieren und anhand des übergebenen Tabs zu reagieren. Dem Tab kann bei seiner Erstellung z.B. ein Tag mitgegeben werden (`actionBar.newTab().setText("Tab 3").setTag(1)` oder `actionBar.newTab().setText("Tab 3").setTag("tab3")` oder ähnlichem), das dann im Listener ausgewertet werden kann.

3.13.6 Animation

Die Übergänge zwischen den Fragmenten in einer Transaktion können mit einer Animation unterlegt werden.

Mit `FragmentTransaction.setTransition(<TRANSITION>)` lässt sich eine Standardanimation für den Übergang in der Transaktion setzen:

FragmentTransaction.TRANSIT_ FRAGMENT_OPEN	Standardanimation beim Öffnen eines Fragments durch `FragmentTransaction.add(…);` Sinnvoll beim Hinzufügen eines Fragments, auch mit vorherigem Entfernen eines bestehenden Fragments. Der Effekt mutet wie ein softes Einblenden an.
FragmentTransaction. TRANSIT_ FRAGMENT_CLOSE	Standardanimation beim Schließen eines Fragments durch `FragmentTransaction.remove(…);` Sinnvoll beim Entfernen oder Verstecken. Weniger sinnvoll wenn direkt ein Fragment in der Transaktion hinzugefügt wird. Der Effekt mutet wie ein softes Ausblenden an.
FragmentTransaction. TRANSIT_ FRAGMENT_FADE	Standardanimation beim Übergang zwischen zwei Fragmenten durch `FragmentTransaction.replace(…);` Die Fragmente werden überblendet.
FragmentTransaction. TRANSIT_ NONE	Keine Animation

Tabelle 3.44: **Fragment Standardanimationen**

Aufwendigere Animationen lassen sich über Property Animations realisieren, **nicht** mit und per `FragmentTransaction.setCustomAnimations(<Resource-ID für die Animation bei Öffnen eines Fragments>, <Resource-ID für die Animation bei Schliessen eines Fragments>)` in einer Transaktion ausführen:

Listing 3.125: **Anwenden einer Property Animation**

```
private void addFragmentWithAnimation2(boolean addToBackStack)
{
    FragmentManager fragmentManager = getFragmentManager();
    FragmentTransaction ta = fragmentManager.beginTransaction();
    ta.setCustomAnimations(R.anim.fragment_enter_animation2, R.anim.frag-
ment_exit_animation2);
    Fragment fragment;
    fragment = LifecycleFragment.createFragment(instanceNo);
    ta.replace(R.id.lifecyclefragment, fragment);
    ta.setBreadCrumbShortTitle("#"+Integer.toString(instanceNo));
    ta.setBreadCrumbTitle("#"+Integer.toString(instanceNo));
    if (addToBackStack) ta.addToBackStack(null);
    ta.commit();
    fragmentManager.executePendingTransactions();
    instanceNo+=1;
}
```

Im obigen Beispiel wird in der Transaktion die Animation gesetzt. Die Animation lässt das schließende Fragment nach rechts heraus- und das öffnende Fragment von links hineingleiten. Die folgenden zwei Listings zeigen die dazugehörige Definition.

Listing 3.126: **Animation res\animator\fragment_enter_animation2.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <objectAnimator
    android:interpolator="@android:anim/accelerate_decelerate_interpolator"
    android:propertyName="alpha"
    android:duration="1500"
    android:valueFrom="0"
    android:valueTo="1"/>
    <objectAnimator
    android:interpolator="@android:anim/accelerate_decelerate_interpolator"
    android:propertyName="translationX"
    android:duration="1500"
    android:valueFrom="-1024"
    android:valueTo="0"/>
</set>
```

Listing 3.127: **Animation res\animator\fragment_exit_animation2.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <objectAnimator
    android:propertyName="alpha"
    android:duration="1500"
    android:valueFrom="1"
    android:valueTo="0"/>
```

```
    <objectAnimator
    android:interpolator="@android:anim/accelerate_decelerate_interpolator"
    android:propertyName="translationX"
    android:duration="1500"
    android:valueFrom="0"
    android:valueTo="1024"/>
</set>
```

Animationseffekte lassen sich so überall dort einsetzen, wo Fragmente angezeigt oder verborgen werden. In Kombination mit dem Property-Animation-Framework lassen sich so vielfältige Effekte erzielen, die die »User Experience« verbessern (manchmal auch verschlimmbessern) können.

TIPP

Die Eigenschaften der Fragmente decken sich natürlich mit denen der View-Basisklasse. Wenn man wissen will, welche Eigenschaften sich zum Animieren eignen, schaut man am besten dort nach.

Es gilt zu beachten, dass hier ausschließlich das Property-Animation-Framework benutzt werden kann. Das View-Animation-Framework funktioniert hier nicht.

3.14 Content-Provider

Die meisten Anwendungen erfassen und speichern Informationen. Das fängt bei den Kontakten und Adressen an, geht über Bilder, Videos und Musik bis hin zu den Highscores von Spielen. Immer wenn wir Informationen erfassen, und sei es lediglich eine Notiz oder eine Einkaufsliste, erfassen wir strukturierte Daten, die meist aus mehreren Bestandteilen und mehreren Einträgen bestehen. Eine Notiz z.B. erhält neben dem Text vielleicht noch einen Betreff, möglicherweise das Datum und die Uhrzeit, wann wir die Notiz angelegt haben, und ruck, zuck können wir uns weitere Dinge vorstellen, die wir zur Notiz speichern wollen. Unsere Kontakte bestehen aus vielen Elementen wie Name, Vorname, Rufname, Spitzname, E-Mail-Adresse, Telefonnummer, Skype-Nummer, Wohnort usw.

Damit wir in unserer Anwendung solche Daten speichern, diese Daten anderen Anwendungen zur Verfügung stellen und selbst auf die Daten anderer Anwendungen zugreifen können, stellt Android das Konzept der Content-Provider bereit. Der Content-Provider stellt definierte Methoden zur Verfügung, um Daten unserer eigenen und anderer Anwendungen abzufragen sowie Daten unserer Anwendung zu speichern und zu verändern und, wenn wir das Recht dazu haben, auch Daten anderer Anwendungen zu erstellen und zu verändern.

Am Beispiel unserer Notiz wäre es z.B. möglich, einen Kontakt aus dem Adressbuch auszuwählen, den wir mit dieser Notiz verbinden, vielleicht weil es sich um eine Anrufnotiz handelt oder weil wir uns erinnern wollen, jemanden anzurufen. In diesem Fall ist es ja sicherlich sinnvoller, auf die Adressen des Adressbuchs zuzugreifen als selbst parallel Namen innerhalb der Notizanwendung zu speichern. Darüber hinaus kann es noch sinnvoll sein, aus der Notizanwendung neue Kontakte anzulegen, falls wir den Kontakt noch gar nicht in unserem Adressbuch erfasst haben.

All das bewerkstelligen wir über Content-Provider.

Android liefert selbst in der Grundausstattung diverse Content-Provider mit. Neben dem angesprochenen Adressbuch sind auch die Anrufliste, die Mediendatenbank (Bilder, Musik) sowie die SMS-Datenbank und auch die Kalenderdatenbank als Content-Provider ausgeführt.

Jeder Content-Provider wird über eine eindeutige Adresse, einen URI angesprochen. Wie bereits bei der Betrachtung des Datenzugriffs aus dem Benutzerinterface heraus besprochen, ist diese Adresse einer Webadresse sehr ähnlich.

Wir erinnern uns, der Aufbau des URI der Content-Provider folgt dem Schema:

```
content://<AUTHORITY>/<TABLE_NAME>
```

Dabei ist `<AUTHORITY>` der »Besitzer« der Daten, z.B. unsere Applikation oder die Bildgalerie, und wird im Allgemeinen aus dem Package-Namen des Content-Providers gebildet, `<TABLE_NAME>` adressiert dann eine konkrete Tabelle innerhalb unseres Content-Providers.

Innerhalb einer Tabelle werden die einzelnen Einträge in Zeilen organisiert, die per Definition immer eine eindeutige ID haben müssen, die in der Spalte _ID verwaltet wird. Um eine Zeile zu adressieren und damit auf einen bestimmten Datensatz zuzugreifen, hängen wir an unseren URI noch die ID an. Einen vollständigen URI zum Zugriff auf eine Zeile sieht dann folgendermaßen aus:

```
content://<AUTHORITY>/<TABLE_NAME>/<ID>
```

Diesem Schema folgen alle Anwendungen, die Content-Provider implementieren. Der Zugriff erfolgt dann über sogenannte Content Resolver. Der Content Resolver stellt Methoden zum Abfragen (Select), Erstellen (Insert), Aktualisieren (Update) und Löschen (Delete) von Daten zur Verfügung. Welche Daten wir adressieren, übergeben wir in den jeweiligen Mehtoden mittels des entsprechenden URI.

Der Resolver löst nun die Adresse auf und sucht den zur `<AUTHORITY>` passenden Content-Provider. Das wird wiederum im Manifest zur Applikation festgelegt. Dort wird konfiguriert, welche Klasse in der Anwendung deren Content-Provider für `<AUTHORITY>` implementiert. Hat der Resolver eine entsprechende Zuordnung gefunden, so wird der Provider gestartet und die entsprechende Methode ausgeführt. Der Provider nun schaut in der Adresse nach, welche Tabelle (`<TABLE_NAME>`) gemeint ist, und führt die gewünschte Operation auf der Tabelle aus, also z.B. eine Abfrage, die Inhalte der Tabelle in Form eines Cursors zurückliefert. Vielleicht ist ja sogar eine `<ID>` angegeben, dann liefern wir einen Cursor zurück, der nur die entsprechende Zeile beinhaltet.

Das heißt, wenn wir einen Content-Provider erstellen, müssen wir auf die Adresse, die unseren Provider adressiert, in unserem Provider reagieren und die Selects, Inserts, Updates und Deletes entsprechend realisieren. Das wiederum erledigen wir mit den Möglichkeiten, die uns die eingebaute SQLite-Datenbank bereitstellt, ggf. aber auch in Verbindung mit Operationen auf dem Dateisystem. Denn: Der Content-Provider ist nicht auf die Speicherung innerhalb der SQ-Lite festgelegt, wir können die Art und Weise des Zugriffs innerhalb des Providers frei wählen.

TIPP

Es ist aber eine gute Idee, für alle Formen von strukturierten Daten die Ablage in der SQLite zu wählen, da wir dann auch Abfragen bequem per SQL ausführen können und das Select-, Insert-, Update-, Delete – Schema perfekt und standardisiert gelöst ist. Wenn wir zu einem Datensatz allerdings noch große Datenmengen, z.B. ein Bild oder eine Voice-Mail o.Ä. speichern wollen, dann bietet es sich an, für diese Daten einen Provider zu schreiben, der die großen Dateien auch wirklich in Dateien speichert und nur einen entsprechenden Verweis in der Datenbank ablegt.

3.14.1 Zugriff auf bestehende Content-Provider

Beschäftigen wir uns zuerst mit dem Zugriff auf bestehende Content-Provider. Eine Form haben wir bei den Cursor-Adaptern schon kennengelernt, wo wir auf das Adressbuch zugegriffen haben, um die Adressen in einer Liste darzustellen:

```
Cursor cur = managedQuery(ContactsContract.Data.CONTENT_URI, PROJECTION,
ContactsContract.Data.MIMETYPE +" = ?", new String[] { android.provider.Con➥
    tactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE }, andro➥
    id.provider.ContactsContract.CommonDataKinds.StructuredName.DISPLAY_
NAME);
```

Hier versteckt sich der Zugriff auf den Content Resolver hinter dem Aufruf von `managedQuery(…)` innerhalb unserer Activity. Das Besondere an `managedQuery(…)` ist, dass die Activity den Lebenszyklus der Abfrage verwaltet und sich darum kümmert, dass beim Deaktivieren der Activity auch der Content Resolver deaktiviert und beim neuerlichen Aktivieren der Activity die Abfrage erneut ausgeführt wird. Wir werden dazu noch einen weiteren Mechanismus kennenlernen, den Android in der Version 3 mit dem Konzept der *Loader* einführt. Mit den *Loadern* kann man nämlich sehr einfach lang laufende Abfragen in den Hintergrund verlagern und damit die Anwendung »geschmeidig« halten, weil sie nicht blockiert, solange die Abfrage läuft.

Hinter dem Aufruf von `managedQuery(…)` steckt im Prinzip der Aufruf von:

```
Cursor cur = getContentResolver().query(ContactsContract.Data.CONTENT_URI,
PROJECTION, ContactsContract.Data.MIMETYPE +" = ?", new String[] { android.
provider.ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE
}, android.provider.ContactsContract.CommonDataKinds.StructuredName.DISPLAY_
NAME);
```

Die `query(…)`-Methode des Content Resolvers, die einen Cursor zurückliefert.

Hinter der Konstante `ContactsContract.Data.CONTENT_URI` steckt die Adresse `content://com.android.contacts/data`.

Der Resolver sucht nun den Provider für die Authority `com.android.contacts`, aktiviert ihn, und der Provider liefert uns Daten aus der Tabelle `data`, passend zur Abfrage die wir an `query(…)` übergeben haben.

INFO

Dadurch, dass die konkrete Implementierung vor uns verborgen bleibt, besagt die Adresse nicht unbedingt dass es eine Tabelle `DATA` gibt, möglicherweise liefert der Provider auch Daten aus unterschiedlichen Tabellen. Gerade das Adressbuch ist extrem flexibel aufgebaut, um die Daten zu einer Person aus unterschiedlichen Quellen wie Facebook, Google Mail, Exchange etc. unter einem einzigen Eintrag zu aggregieren. Die Tabelle `DATA` ist tatsächlich eine Tabelle, deren Zeilen allerdings nicht genau einen Kontakt ausmachen, sondern ein Kontakt ist ein Aggregat aus verschiedenen Zeilen dieser allgemeinen Tabelle `DATA`. An dieser Tabelle `DATA` könnte man ansetzen, um in einer eigenen Anwendung zusätzliche Informationen an einen Kontakt zu hängen.

Schauen wir uns die wichtigsten Methoden des Content Resolvers an:

ContentProviderResult[]	applyBatch(String authority, ArrayList<ContentProviderOperation> operations)	Ausführen eines Stapels an Operationen. Im Array operations werden die entsprechenden Operationen (Insert und/oder Update) übergeben und dann »in einem Rutsch« ausgeführt. Im Gegensatz zu den dezidierten Operationen werden die Batches auf der AUTHORITY, d.h. dem Content-Provider, ausgeführt, und jede einzelne Operation im Batch kann verschiedene URIs bzw. Tabellen innerhalb des Providers ansprechen.
final int	bulkInsert(Uri url, ContentValues[] values)	Mehrere neue Zeilen (Datensätze) in einer Tabelle bzw. an der Adresse anlegen.
final int	delete(Uri url, String where, String[] selectionArgs)	Zeilen/Datensätze löschen
final Uri	insert(Uri url, ContentValues values)	Eine einzelne neue Zeile anlegen
final Cursor	query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)	Datensätze abfragen. Das Abfrageergebnis wird als Cursor zurückgegeben, mit dem die einzelnen Datensätze abgerufen werden können.
final int	update(Uri uri, ContentValues values, String where, String[] selectionArgs)	Eine oder mehrere Datensätze/ Zeilen aktualisieren.

Tabelle 3.45: **Die wichtigsten Operationen auf einem Content Resolver**

final void	registerContent-Observer(Uri uri, boolean notifyForDescendents, ContentObserver observer)	Registrieren eines »Beobachters«, der über Änderungen an den Datensätzen/Zeilen bzw. dem Inhalt der Adresse informiert wird.
final void unregisterContentObserver(ContentOb server observer)		Entfernen eines Beobachters.

Tabelle 3.45: **Die wichtigsten Operationen auf einem Content Resolver (Forts.)**

Der Content Resolver bietet noch weitere Methoden bezüglich des Datenabgleichs an, die wir uns aber erst in diesem Zusammenhang anschauen werden.

Die hier vorgestellten Methoden dienen nun in erster Linie dazu, Datensätze zu bearbeiten und abzufragen.

TIPP

Wir sollten diese Mechanismen auch für unsere eigenen Content-Provider benutzen und nicht innerhalb der Anwendung direkt auf die Datenbank zugreifen, auch wenn es sich um unsere eigene handelt. Wenn wir also in unserem Scrapbook neue Einträge erstellen wollen, dann sprechen wir unseren Content-Provider genauso an, als wäre es der Provider einer anderen Anwendung. Dadurch entkoppeln wir das Datenmanagement sehr sauber von der Anwendung, die die Daten benutzt.

Wie wir bereits besprochen haben, sind die Daten in der Regel tabellenartig organisiert, das heißt, die Informationen sind in Zeilen und Spalten abgelegt.

Neben dem Content-URI, der die Ablage bzw. Tabelle adressiert, in der wir Daten aufheben wollen, müssen wir dann also auch die Namen der Spalten wissen, in denen wir die einzelnen Elemente unserer Daten unterbringen möchten.

Die Namen der Spalten werden, genau wie der Content-URI, durch die jeweiligen Implementierungen innerhalb der Content-Provider festgelegt und in der Regel als Zeichenketten-Konstanten in den jeweilgen Klassen veröffentlicht.

Damit sind wir an dem Punkt angelangt an dem es sinnvoll ist, sich mit den standardmäßig vorhandenen Content-Providern auseinanderzusetzen. Um auf die Kontakte, die Anrufliste, die Medienbibliothek etc. zuzugreifen, muss man wissen, wie die Content-URIs lauten und welche Spalten die jeweiligen Tabellen besitzen.

Die bekannten Provider liegen zum größten Teil innerhalb des Packages `android.provider`. Die offizielle Dokumentation listet diese auf, verbirgt jedoch einige interessante Provider wie die SMS-Datenbank und die Kalenderdatenbank. Das ist wohl dem Umstand geschuldet, dass manche Datenbanken noch sehr stark dem evolutionären Entwicklungsprozess unterworfen sind und sich noch verändern können, so dass umfangreichere Programmanpassungen notwendig werden. Über Google Code Search (`http://www.google.com/codesearch`) kann man aber in den aktuellen Android-Source-Code einsteigen und auch in den nicht offiziell dokumentierten Providern stöbern um Content-URIs und Spaltennamen herauszufinden.

PROVIDER	ZWECK	BENÖTIGTES RECHT
Browser Browser.BOOKMARKS_URI Browser.BookmarkColumns	Bookmarks und Historie	READ_HISTORY_ BOOKMARKS WRITE_HISTORY_ BOOKMARKS
CallLog.Calls CallLog.Calls.CONTENT_URI	Anrufliste	
ContactsContract ContactContracts.Contacts. CONTENT_URI ContactContracts.RawContacts. CONTENT_URI	Kontakte, und zwar »rohe« Kontakte (aus unterschiedlichen Quellen) sowie aggregierte Kontakte (die aus unterschiedlichen Quellen zu einer Person gehören)	READ_CONTACTS WRITE_CONTACTS
MediaStore android.provider.MediaStore. MediaColumns MediaStore.Audio.Media. EXTERNAL_CONTENT_URI MediaStore.Audio.Media. INTERNAL_CONTENT_URI android.provider.MediaStore. Audio.AudioColumns MediaStore.Images.Media. EXTERNAL_CONTENT_URI MediaStore.Images.Media. INTERNAL_CONTENT_URI android.provider.MediaStore. Images.ImageColumns MediaStore.Videos.Media. EXTERNAL_CONTENT_URI MediaStore.Videos.Media. INTERNAL_CONTENT_URI android.provider.MediaStore. Videos.VideoColumns	Audio, Bilder und Videos. Wichtig ist die Unterscheidung zwischen den EXTERNAL_- und den INTERNAL_-Content-URIs. EXTERNAL_ adressiert immer die Medien, die nicht zu den systemeigenen Medien gehören, und hat nichts damit zu tun, ob die Medien auf einer externen Speicherkarte liegen.	
UserDictionary.Words UserDictionary.Words. CONTENT_URI	Einträge im Benutzerwörterbuch	

Tabelle 3.46: **Offiziell dokumentierte Content-Provider**

Eine denkbare, sehr sinnvolle Anwendung wäre z.B. das Bearbeiten der Informationen zu den Bildern in unserer Galerie. Die Standardanwendungen, die mit den Geräten ausgeliefert werden, halten sich an dieser Stelle ziemlich bedeckt und bieten kaum Verwaltungsmöglichkeiten.

In/auf der Spielwiese haben wir im Zusammenhang mit den Fragmenten eine Activity `ShowImagesWithFragments` erstellt, die uns die Bilder aus dem MediaStore auflistet und es erlaubt, zu einem Bild den Titel und die Beschreibung zu bearbeiten.

Das Bearbeiten ist in eine eigene Activity `EditMediaActivity` ausgelagert, die, wie es die Modularisierung vorsieht, über ein Intent gestartet wird, auch wenn wir die Activity in unserer eigenen Anwendung verwenden. Diese Activity wird in der Spielwiese z.B. auch dazu benutzt, für ein von der Kamera aufgenommenes und abgespeichertes Bild den Titel und die Beschreibung direkt nach der Aufnahme zu bearbeiten.

Innerhalb des Fragments `ShowImagesFragment`, das eine Listview mit Thumbnails der Bilder darstellt, wird bei Auswahl eines Eintrags das Fragment `ShowImageFragment` mit dem URI des ausgewählten Bildes aktualisiert:

Listing 3.128: **Auswahl eines Eintrags in der Bilderliste**

```
void showDetails(int position)
{
    if (position<getListView().getCount())
    {
        currentSelection = position;
        long id = getListView().getItemIdAtPosition(position);
        Uri uri = ContentUris.withAppendedId(contentUri, id);
        if (hasDualPane)
        {
        getListView().setSelection(position);
        getListView().setItemChecked(position, true);
        Fragment detailFragment = getFragmentManager().findFragmentById(R.
id.showimagefragment);
        if (detailFragment!=null)
        {
        ((ShowImageFragment)detailFragment).setImageURI(uri);
        }
    }
    else
    {
        startActivity(new Intent(getActivity(),ShowImageActivity.class).
putExtra("data",uri.toString()));
        }
    }
}
```

Wichtig ist hier das Konstrukt:

```
long id = getListView().getItemIdAtPosition(position);
Uri uri = ContentUris.withAppendedId(contentUri, id);
```

Wobei gilt:

```
contentUri = MediaStore.Images.Media.EXTERNAL_CONTENT_URI;
```

Per Konvention müssen Content-Provider für jeden Datensatz eine eindeutige, unveränderliche ID bereitstellen (was auf SQLite-Datenbanken mit einer Auto-Increment-Spalte als Primärschlüssel erreicht wird). Diese ID wird mittels `getItemIdAtPosition(…)` ermittelt und an den Basis-URI des MediaStores angehängt. Dadurch wird dann ein Eintrag in der Tabelle eindeutig adressiert.

Diese eindeutige Adresse wird mit

```
((ShowImageFragment)detailFragment).setImageURI(uri);
```

an das Fragment mit der Detailansicht des Bildes übergeben. Damit kennt diese Detailansicht die Adresse des Bildes und kann es darstellen:

Listing 3.129: **Anzeigen des Bildes im Detail-Fragment**

```
public void setImageURI(Uri uri)
{
    if (uri.equals(imageUri))
    {
        return;
    }

    imageUri = uri;

    ImageView imageView = (ImageView)this.getView().findViewById(R.
id.showimage_imageview);
    if (imageView!=null)
    {
        InputStream is;
        try {
        if (bitmap!=null)
        {
        bitmap.recycle();
        }
        is = getActivity().getApplication().getContentResolver().
openInputStream(uri);
        bitmap = BitmapFactory.decodeStream(is);
        is.close();
        imageView.setImageBitmap(bitmap);
        } catch (FileNotFoundException e) {
        } catch (IOException e) {
        }
    }
}
```

Der essenzielle Aufruf hier ist:

```
is = getActivity().getApplication().getContentResolver().
openInputStream(uri);
```

um einen Zugriff auf die Bilddaten zu erhalten. Hier liefert der Content Resolver also keinen Cursor auf Datensätze zurück, sondern einen `InputStream` auf die Bilddaten (die auf dem Dateisystem gespeichert sind, das muss aber nicht so sein!).

TIPP

Das ist für alle Medieninhalte des MediaStores so gelöst und eine schicke Sache. Der Content Resolver wird damit nach außen hin die zentrale Instanz zum Zugriff auf alle Daten im MediaStore, egal ob Metadaten in der Datenbank oder die eigentlichen Mediendaten. Wir sollten das, falls unsere Anwendung so etwas auch macht, ebenso realisieren. Der Nutzer des Content-Providers muss sich dann keine Gedanken darüber machen, wo wir unsere Dateien speichern.

Das Optionenmenü des Detailfragments erlaubt nun das Bearbeiten der Metadaten.

Listing 3.130: **Starten einer Activity zum Bearbeiten der MediaStore-Daten**

```
@Override
public boolean onContextItemSelected(MenuItem item) {
        switch(item.getItemId())
        {
        case R.id.item_edit:
        startActivity(new Intent(Intent.ACTION_EDIT,imageUri));
        break;
        }
        return super.onContextItemSelected(item);
}
```

Der Aufruf von

```
startActivity(new Intent(Intent.ACTION_EDIT,imageUri));
```

benutzt ein Intent mit der Aktion `Intent.ACTION_EDIT`, und zwar soll diese Aktion auf den Daten hinter `imageUri` ausgeführt werden. Das ist die allgemeinste Form, eine Aktion auf Daten auszuführen. Das System sucht nun nach Activities, die auf diesen Daten ein `ACTION_EDIT` ausführen können, und bringt diese dann zur Ausführung. Gibt es mehrere Activities, kann der Anwender aus diesen wählen. DAS ist Modularisierung.

Unsere Activity ist im Manifest entsprechend definiert:

Listing 3.131: **Intent-Filter für die EditMediaActivity**

```
<activity android:theme="@android:style/Theme.Translucent.NoTitleBar"
android:name="de.androidpraxis.SpielwieseLibrary3.EditMediaActivity">
    <intent-filter>
        <category android:name="android.intent.category.DEFAULT"></category>
        <action android:name="android.intent.action.EDIT"></action>
        <data android:mimeType="image/*" android:scheme="content"/>
    </intent-filter>
</activity>
```

Der Content-Provider liefert für den Content-URI eines Eintrags für die Bildergalerie immer image/* (z.B. image/jpeg oder image/png, je nachdem, welches Format das Bild hat). Wir setzen also hier den Intent-Filter auf diesen mimeType, beschränken das Ganze aber auf Quellen aus den Content-Providern.

Damit kommt unsere Activity für ACTION_EDIT infrage und wird ausgeführt.

Der Zugriff auf die Metadaten erfolgt dort ebenfalls über den Content Resolver, denn das Intent transportiert ja genau die Adresse der Daten, die wir bearbeiten wollen:

Listing 3.132: **Zugriff auf die Daten**

```
public void mapDataToView()
{
[…]
Intent intent = getIntent();
Uri data = intent.getData();
Cursor cursor = getContentResolver().query(data, new String[] { MediaSto-
re.Images.ImageColumns.TITLE,MediaStore.Images.ImageColumns.DESCRIPTION },
null, null, null);
if (cursor.moveToFirst())
{
    EditText title = (EditText)dialogView.findViewById(R.id.media_title);
    EditText description = (EditText)dialogView.findViewById(R.id.media_de➥
       scription);
    if (title!=null) title.setText(cursor.getString(0));
    if (description!=null) description.setText(cursor.getString(1));
}
[…]
}
```

Entscheidend ist hier der URI, der über das Intent transportiert wird. Per `getContentResolver().query(data,…)` fordern wir über den Content Resolver einen Cursor auf diese Daten an.

Innerhalb unserer Activity können wir nun die Werte bearbeiten. Beim Speichern passiert dann Folgendes:

Listing 3.133: **Aktualisieren des Eintrags**

```
public void mapDataFromView()
{
[…]
Intent intent = getIntent();
Uri data = intent.getData();
ContentValues values = new ContentValues();
EditText title = (EditText)dialogView.findViewById(R.id.media_title);
EditText description = (EditText)dialogView.findViewById(R.id.media_descrip➥
    tion);
if (title!=null) values.put(MediaStore.Images.ImageColumns.TITLE, title.
getText().toString());
if (description!=null) values.put(MediaStore.Images.ImageColumns.DESCRIP➥
    TION, description.getText().toString());
getContentResolver().update(data, values, null, null);
[…]
}
```

Wir ermitteln wieder den URI der Daten und füllen das Objekt `values` vom Typ `ContentValues` mit den Spaltennamen und den Werten der zu aktualisierenden Spalten. Der Aufruf von `getContentResolver().update(`**data**, **values**, `null, null)` schließlich übergibt die Werte an den entsprechenden Content-Provider, die Metadaten werden aktualisiert.

ACHTUNG

Beim obigen Aufruf von `update` **lassen wir das Abfragekriterium weg. Das können wir deswegen machen, weil** `data` **einen einzelnen Datensatz adressiert. Hier ist aber Vorsicht geboten! Adressiert** `data` **nicht einen einzelnen Datensatz, sondern eine ganze Tabelle, dann werden alle Datensätze entsprechend aktualisiert. Das Gleiche gilt auch für den Aufruf von** `delete`**.**

Um sich dagegen zu schützen, gibt es zwei Möglichkeiten:

1. Ermitteln ob der URI eine ID besitzt und somit einen einzigen Datensatz adressiert
2. Erfragen des Typs zur URI vom Content-Provider

Die erste Möglichkeit nutzt die Klasse `ContentUris`:

Listing 3.134: **Ermitteln ob der Content-URI einen einzelnen Datensatz adressiert**

```
public boolean addressesSingleItem(Uri data)
{
    try
    {
        long id = ContentUris.parseId(data);
        return id>=0;
    }
    catch (UnsupportedOperationException e)
    {
        return false;
    }
    catch (NumberFormatException e)
    {
        return false;
    }
}
```

Die zweite Möglichkeit nutzt aus, dass per Definition ein Content-Provider für ein Datenverzeichnis, also wenn eine Tabelle adressiert wird, den Inhaltstyp "*vnd.android.cursor.dir/**<Verzeichnisname/Tabellenname>***" und für einen Datensatz den Inhaltstyp "*vnd.android.cursor.item/**<Verzeichnisname/Tabellenname>***" zurückliefern soll (MediaStore bricht damit ein wenig, da ein Datensatz z.B. den Typ "*image/<bildtyp>*" für den Datensatz zurückliefert).

Listing 3.135: **Ermitteln ob der Content-URI ein Verzeichnis/eine Tabelle adressiert.**

```
public boolean addressesDirectory(Uri data)
{
    String type = getContentResolver().getType(data);
    boolean isDirectory = type.startsWith("vnd.android.cursor.dir/");
    return isDirectory;
}
```

Noch robuster wird das Ganze, wenn wir die erste Möglichkeit noch damit kombinieren, in unserem Update-Statement eine WHERE-Klausel zu kodieren:

Listing 3.136: **Robustes Update**

```
try
{
    long id = ContentUris.parseId(data);
    getContentResolver().update(data, values, MediaStore.ImageColumns._
ID+"=?", new Strings[] { Long.toString(id) });
}
catch (UnsupportedOperationException e)
{
}
catch (NumberFormatException e)
{
}
```

TIPP

Auch hier sei wieder auf die Nutzung von Platzhaltern in der WHERE-Klausel hingewiesen. Das Zusammenbauen in der Form `MediaStore.ImageColumns._ID+"="+ Integer.toString(id)` sollten wir gar nicht erst anfangen! Zu groß sind die Gefahren ungewollter SQL-Injections.

3.14.2 Erstellen eines eigenen Content-Providers

Wir haben nun die Grundlagen und Funktionsweise von Content-Providern und des Content Resolvers kennengelernt, jetzt wollen wir natürlich dieses tolle Stück Funktionalität auch selbst verwenden. Mir ist bei der Beschäftigung mit der Kontaktverwaltung aufgefallen, dass es keine Möglichkeit gibt Gesprächsnotizen mit einem Kontakt zu verknüpfen. Wir wollen im Folgenden einen Content-Provider für Notizen im Allgemeinen erstellen, die aber als Gesprächsnotiz auch mit einem Kontakt verknüpft werden werden können.

Wir legen in der Eclipse also eine neue Klasse *Notizen* an und leiten diese von Content-Provider ab. Wenn wir das direkt mit dem Assistenten erledigen, bereitet Eclipse auch die Methoden vor, die wir dann selbst ausfüllen müssen. Allerdings benennt der Assistent die Argumente teilweise nicht sonderlich sprechend (*arg0…arg<n>*), das sollten wir dann manuell korrigieren:

Listing 3.137: **Rahmen für unseren eigenen Content-Provider**

```
public class SpielwieseProvider extends ContentProvider {
    public int delete(Uri uri, String selection, String[] selectionargs) {
    return 0;
    }

    @Override
    public String getType(Uri uri) {
    return null;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
    return null;
    }
```

```
    @Override
    public boolean onCreate() {
    return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionargs,
    String order) {
    return null;
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
String[] selectionargs) {
    return 0;
    }
}
```

Wir erkennen in der erstellten Klasse einige der Methoden wieder, die wir bei der Benutzung des Content-Resolvers bereits kennen gelernt haben. Diese gilt es nun entsprechend zu füllen.

Als Erstes legen wir aber die AUTHORITY für unseren Content-Provider fest. Potenziell verwaltet der Provider ja nicht nur ein Verzeichnis bzw. eine Tabelle, sondern mehrere. Deshalb haben wir den Provider auch nicht NotizenProvider genannt, denn damit wären wir möglicherweise schon zu sehr festgelegt.

Die AUTHORITY ist auch der Bezeichner, unter dem der Content-Provider im Manifest deklariert wird, und die AUTHORITY bildet sozusagen die Wurzel für die jeweiligen Content-URIs.

Da die URIs und somit auch die AUTHORITY eindeutig sein müssen, ist es sinnvoll, unser Anwendung-Package entsprechend als AUTHORITY zu benutzen:

Listing 3.138: **Deklaration der AUTHORITY**

```
public class SpielwieseProvider extends ContentProvider {
public static String AUTHORITY = "de.androidpraxis.spielwiese3.spielwiese➥
    provider";
[…]
}
```

Listing 3.139: **Deklaration des Providers im Manifest**

```
<provider
android:authorities="de.androidpraxis.spielwiese3.spielwieseprovider"
android:name="de.androidpraxis.SpielwieseLibrary3.provider.SpielwieseProvi➥
    der">
</provider>
```

Als Nächstes legen wir Klassen (eine Klasse) für die Notizen an. Diese Klasse repräsentiert die Tabelle und die Spalten und führt auch den Content-URI ein, unter dem die Notizen später verwaltet werden:

Listing 3.140: **Klasse, die die Notiz(en) repräsentiert**

```
public class Notizen {

public static final int directoryId = 1;
public static final int itemId = 2;

public static final String TABLE_NAME = "notizen";

public static final Uri CONTENT_URI =       Uri.parse("content://"+Spielwie
seProvider.AUTHORITY+"/"+TABLE_NAME);

public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.spiel➥
    wiese.notiz";
public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.
spielwiese.notiz";

public static final String DEFAULT_SORT_ORDER = "datum desc";

    public static class Columns implements BaseColumns {
        public static final String NOTIZ_DATUM = "datum";
        public static final String NOTIZ_BETREFF = "betreff";
        public static final String NOTIZ_TEXT = "text";
        public static final String NOTIZ_CONTACT_ID = "kontakt";
    }
}
```

Wir orientieren uns dabei an der Vorgehensweise, die auch die systemeigenen Provider nutzen.

Die jeweilige Klasse führt den Content-URI ein, der auf der AUTHORITY des Providers basiert, sowie Konstanten für den CONTENT_TYPE und CONTENT_ITEM_TYPE .

INFO

Die Bezeichnungen bzw. Konstanten AUTHORITY und CONTENT_URI sowie CONTENT_TYPE und CONTENT_ITEM_TYPE sind willkürlich gewählt und könnten auch anders heißen. Es ist aber eine stillschweigende Übereinkunft, diese Nomenklatur zu verwenden.

Wofür sind die einzelnen Konstanten nun gut? Wie wir bereits gesehen haben, dient die Content-URI zur Identifikation der Tabelle oder eines Datensatzes. Der Content-Provider muss also die Content-URIs, die an ihn gerichtet sind, irgendwie einer Tabelle zuordnen. Ausserdem muss er auch Auskunft über die Inhaltstypen zu einem URI geben können, wenn diese z.B. über Intent-Filter abgefragt werden. Das findet ja alles im Provider statt, und hier benutzen wir dann die Konstanten.

Außerdem wollen wir ja selbst den Provider nutzen, ähnlich wie in den vorigen Beispielen die systemseitigen Provider. Dafür benötigen wir ebenfalls den URI und Spaltennamen etc.

Um im Provider die URIs zu unterscheiden, bietet sich der UriMatcher an. Diesen Matcher konfigurieren wir mit den möglichen Content-URIs und verknüpfen jeweils eine ID damit, um später sehr einfach zischen den einzelnen URIs unterscheiden zu können:

Listing 3.141: **Konfigurieren des UriMatchers**

```
sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
sUriMatcher.addURI(AUTHORITY,Notizen.TABLE_NAME,Notizen.directoryId);
sUriMatcher.addURI(AUTHORITY,Notizen.TABLE_NAME+"/#",Notizen.itemId);
```

Der erste URI adressiert die Tabelle Notizen, der zweite URI einen Eintrag darin. Das Hash-Zeichen (#) steht hier als Platzhalter für die ID, die einem Content-URI mitgegeben wird, wenn genau ein Datensatz adressiert werden soll.

Wir können einen gegebenen URI dann in folgender Weise »matchen«:

Listing 3.142: **Herausfinden, ob der Content-URI zu uns gehört und was adressiert wird**

```
@Override
public String getType(Uri uri) {
    switch (sUriMatcher.match(uri)) {
        case Notizen.directoryId:
        return Notizen.CONTENT_TYPE;

        case Notizen.itemId:
        return Notizen.CONTENT_ITEM_TYPE;

        default:
        return null;
    }
}
```

Damit wird es für uns sehr einfach herauszufinden, was ein Nutzer unseres Providers adressiert.

Wenn unser Provider nun mehrere Tabellen verwaltet, gehen wir für jede Tabelle nach diesem Schema vor und müssen nur darauf achten, die IDs entsprechend eindeutig hochzuzählen.

Um überhaupt die Daten in der Datenbank verwalten zu können, brauchen wir eine was? Genau, eine Datenbank. Das Framework liefert uns zum Glück alle Mechanismen, mit der SQLite-Datenbank aus unserer Anwendung heraus zu arbeiten. Eine besondere Hilfsklasse ist der `SQLiteOpenHelper`. `SQLiteOpenHelper` bietet uns die beiden Methoden `getReadableDatabase()` und `getWriteableDatabase()` um innerhalb unseres Providers ein geöffnetes und zum Lesen oder Schreiben bereitetes `SQLiteDatabase`-Objekt zu erhalten.

Der Clou ist dabei, dass der Helper sich darum kümmert, die Datenbank anzulegen, falls sie noch nicht existiert, und sich um das Öffnen und Verwalten der Datenbankverbindung kümmert, wenn wir Zugriff auf die Datenbank benötigen. Natürlich kann der Helper nicht wissen, wie unsere Datenbank aussieht, das müssen wir ihm schon sagen.

Listing 3.143: **Erstellen eines SQLiteOpenHelpers**

```
public class SpielwieseProvider extends ContentProvider {
private static class DatabaseHelper extends SQLiteOpenHelper {
private static final String DATABASE_NAME = "spielwiese.db";
private static final int DATABASE_VERSION = 1;
DatabaseHelper(Context context) {
super(context, DATABASE_NAME, null, DATABASE_VERSION);
}
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE " + Notizen.TABLE_NAME + " ("
        + Notizen.Columns._ID + " INTEGER PRIMARY KEY,"
        + Notizen.Columns.NOTIZ_TEXT + " TEXT,"
        + Notizen.Columns.NOTIZ_BETREFF + " TEXT,"
        + Notizen.Columns.NOTIZ_DATUM + " INTEGER,"
        + Notizen.Columns.NOTIZ_CONTACT_ID + " INTEGER"
        + ");");
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
    }
}
[…]
private DatabaseHelper databaseHelper;
@Override
public boolean onCreate() {
    databaseHelper = new DatabaseHelper(getContext());
    return true;
    }
    […]
}
```

Wichtig sind die Methoden `onCreate(…)` und `onUpgrade(…)`, die aufgerufen werden könnten, sobald wir Zugriff auf die Datenbank anfordern. In `onCreate(…)` erstellen wir die Datenbank, und in `onUpgrade(…)` können wir bei einem Versionswechsel alles dafür tun, die Datenbank ebenfalls zu aktualisieren. Android verwaltet die Versionsnummer, und sobald wir in einer neuen Version unserer Anwendung die Konstante `DATABASE_VERSION` von 1 auf 2 ändern, ruft der Helper die `onUpgrade(…)`-Methode auf. Natürlich müssen wir dann das Entsprechende tun, um unsere Datenbank von 1 nach 2 zu bringen, ggf. aber auch von 1 nach 4, je nachdem, wie lange unser Anwender nicht mehr die Anwendung aktualisiert hat.

Der Zugriff auf die Datenbank erfolgt dann über `SQLiteDatabase db = databaseHelper.getReadableDatabase()` respektive `SQLiteDatabase db = databaseHelper.getWritableDatabase()`, je nachdem, ob wir nur eine Abfrage oder ein Insert, Update oder Delete ausführen wollen.

Gut, schauen wir uns an was wir noch so alles anstellen müssen. Als Erstes wollen wir Anfragen nach den Notizen bedienen:

Listing 3.144: **Abfrage ausführen**

```
public Cursor query(Uri uri, String[] projection, String selection, String[]
selectionArgs,
String order) {

    Cursor result = null;
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
    String orderBy = order;

    switch (sUriMatcher.match(uri)) {

        case Notizen.directoryId:
        qb.setTables(Notizen.TABLE_NAME);
        if (TextUtils.isEmpty(orderBy)) orderBy = Notizen.DEFAULT_SORT_OR➥
    DER;
        break;

        case Notizen.itemId:
        qb.setTables(Notizen.TABLE_NAME);
        String id = uri.getLastPathSegement();
        qb.appendWhere(Notizen.Columns._ID+"="+id);
        break;

        default:
        throw new IllegalArgumentException("Unknown URI " + uri);
    }

    result = qb.query(databaseHelper.getReadableDatabase(), projection, se➥
      lection, selectionArgs, null, null, orderBy);
    result.setNotificationUri(getContext().getContentResolver(), uri);
    return result;
}
```

Auch hier wird wieder per UriMatcher unterschieden, was abgefragt werden soll. Wenn eine ID in dem Content-URI angegeben wurde, dann erzeugen wir eine entsprechende Abfrage auf diese ID, wenn die Tabelle abgefragt wird, wird einfach die Originalabfrage ohne veränderung ausgeführt.

Mit dem Aufruf von `result.setNotificationUri(getContext().getContentResolver(), uri)` erhält der Cursor noch den Hinweis, auf welchen Content-URI er für Änderungen an den zugrunde liegenden Daten horchen soll, um sich ggf. zu aktualisieren. Das ist ein klassisches Observer-Muster. Wir müssen in unserem Provider Änderungen an den Daten entsprechend signalisieren.

Spannend ist jetzt natürlich noch, wie wir uns im Falle von Insert, Update und Delete verhalten.

Es ist offensichtlich dass ein Insert nur auf die Tabelle stattfinden kann, nicht auf einen Eintrag selbst – es sein denn, wir implementieren eine hierarchische Semantik, bei der z.B. untergeordnete Datensätze über den URI des übergeordneten Datensatzes eingehängt werden. Davon sehen wir hier aber ab, da es in unserem Beispiel keinen Sinn ergibt.

Here we go:

Listing 3.145: **Insert**

```
@Override
public Uri insert(Uri uri, ContentValues initialValues) {
    if (sUriMatcher.match(uri)!=Notizen.directoryId)
    {
        throw new IllegalArgumentException("Unknown URI " + uri);
    }
    SQLiteDatabase db = databaseHelper.getWritableDatabase();

    ContentValues values;
    if (initialValues != null) {
        values = new ContentValues(initialValues);
    } else {
        values = new ContentValues();
    }
    Long now = Long.valueOf(System.currentTimeMillis());
    if (!values.containsKey(Notizen.Columns.NOTIZ_DATUM))
    {
        values.put(Notizen.Columns.NOTIZ_DATUM,now);
    }
    long rowId = db.insert(Notizen.TABLE_NAME, Notizen.Columns.NOTIZ_BE-
TREFF, values);
    if (rowId > 0) {
        Uri noteUri = ContentUris.withAppendedId(Notizen.CONTENT_URI, row➥
    Id);
        getContext().getContentResolver().notifyChange(noteUri, null);
        return noteUri;
    }
    throw new SQLException("Failed to insert row into " + uri);
}
```

Unsere Insert-Methode stellt erst einmal sicher, dass auch das aktuelle Datum für die Datum-Spalte gesetzt wird. Das Interessante ist der Insert-Aufruf selbst, denn dieser liefert die neu erzeugte ID des Datensatzes zurück.

INFO

Deshalb muss jede Tabelle eine Auto-Increment-Spalte als Primärschlüssel mit dem Namen »_ID« besitzen. Damit kann die eindeutige ID erzeugt und zurückgegeben werden.

Nur so ist es möglich, einen eindeutigen URI zu diesem Datensatz zu erstellen.

Auf höherer Ebene (AdapterViews) wird diese ID ebenfalls benötigt.

Ebenfalls bemerkenswert ist der Aufruf von `notifyChange(…)`. Das ist das Signal für alle Beobachter des Content-URI, dass sich etwas geändert hat. Die Adapter z.B. werden sich dann aktualisieren, um die Änderungen auch zeitnah zu reflektieren.

Listing 3.146: **Update**

```
@Override
public int update(Uri uri, ContentValues values, String where, String[]
whereArgs) {
    SQLiteDatabase db =databaseHelper.getWritableDatabase();
```

```
    int count;
    switch (sUriMatcher.match(uri)) {
    case Notizen.directoryId:
        count = db.update(Notizen.TABLE_NAME, values, where, whereArgs);
    break;

    case Notizen.itemId:
        String noteId = uri. getLastPathSegement();
        count = db.update(Notizen.TABLE_NAME, values, Notizen.Columns._ID +
"=" + noteId
        + (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""), where-
Args);
    break;
    default:
    throw new IllegalArgumentException("Unknown URI " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}
```

Das Update liefert die Anzahl der geänderten Zeilen zurück. Die IDs selbst verändern sich nicht. Hier ist wieder das Augenmerk darauf zu richten, dass im Falle eines eindeutigen URI die Abfrage auf die eindeutige ID noch in die Where-Klausel eingeflochten wird. Man könnte nun argumentieren, dass doch das übrige *where* in diesem Moment überflüssig wäre. Dem ist mitnichten so, denn es kann ja durchaus sein, dass noch ein weiteres Kriterium stimmen muss, um das Update durchzulassen, z.B. »READ_ONLY is NULL«, wenn unsere Datensätze vor Überschreiben geschützt werden sollen.

Listing 3.147: **Delete**

```
@Override
public int delete(Uri uri, String where, String[] whereArgs) {
    SQLiteDatabase db = databaseHelper.getWritableDatabase();
    int count;
    switch (sUriMatcher.match(uri)) {
        case Notizen.directoryId:
        count = db.delete(Notizen.TABLE_NAME, where, whereArgs);
        break;
        case Notizen.itemId:
        String noteId = uri.getLastPathSegement();
        count = db.delete(Notizen.TABLE_NAME, Notizen.Columns._ID + "=" +
noteId
        + (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""), where➥
    Args);
        break;
        default:
        throw new IllegalArgumentException("Unknown URI " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}
```

Auch das Delete verhält sich »straight forward«. Ähnlich wie beim Update liefert es die Anzahl der betroffenen Zeilen zurück und bastelt im Falle der eindeutigen URI noch die ID in die Abfrage mit ein.

Auf diese Weise können wir viele verschiedene Datenbanktabellen durch unseren Content-Provider managen lassen. Es ist offensichtlich, dass aber mit zunehmender Anzahl der Tabellen dieser Ansatz schnell unübersichtlich wird.

TIPP

Flexibler können wir das Ganze gestalten, wenn wir unsere Klassen wie Notizen etc. als Prototypen aufbauen. Diesen Ansatz möchte ich hier kurz skizzieren.

Auf der Spielwiese, genauer in der `SpielwieseLibrary3`, befinden sich die Klasse `DatabaseProvider` und das Interface `DatabaseTable`. `DatabaseTable` definiert alle Methoden, die der `ContentProvider` benötigt um, mit einer Tabelle zu arbeiten. Unter anderem liefert eine Implementierung den Tabellennamen, den Content-URI, den Inhaltstyp und kümmert sich um das Insert, Update und Delete.

In einer konkreten Ableitung des DatabaseProvider kann man die jeweiligen Implementierungen als Prototypen zu dem Provider zufügen, dieser kümmert sich intern um das Matching und auch darum, ob es sich um einen URI mit ID handelt.

Der Provider selbst wird somit recht allgemeingültig, eine Ableitung muss nur den Datenbanknamen und die Datenbankversion bereitstellen sowie die DatabaseTable-Implementierungen zufügen.

Die eigentliche Spezialisierung findet dann in der Implementierung der DatabaseTable statt, die allerdings auch ziemlich schlank und »straight forward« ist.

Unser `ContentProvider` schrumpelt mit diesem Ansatz extrem zusammen:

Listing 3.148: **Einsatz eines prototypbasierten Ansatzes**

```
public class SpielwieseProvider2 extends DatabaseProvider {
    public static String AUTHORITY = "de.androidpraxis.spielwiese3.spielwie➥
    seprovider";

    @Override
    protected String getAuthority() {
    return AUTHORITY;
    }

    @Override
    protected String getDatabaseName() {
    return "spielwiese.db";
    }

    @Override
    protected int getDatabaseVersion() {
    return 1;
    }
```

```
    @Override
    public void addTables() {
    add(new NotizenPrototype());
    }
}
```

INFO

Der `DatabaseProvider` ist von `ContentProvider` abgeleitet und implementiert die allgemeinen Aufgaben im Zusammenhang mit der datenbankbasierten Speicherung.

Die Notizen-Implementierung nutzt die bereits bestehende Notizen-Klasse (die man als Client-Interface bezeichnen könnte) und reichert sie einfach um die Methoden des Prototypen an:

Listing 3.149: **Der Notizen-Prototyp**

```
public class NotizenPrototype extends Notizen implements DatabaseTable {
    @Override
    public String getTableName() {
    return TABLE_NAME;
    }
    @Override
    public String getDefaultOrder() {
    return DEFAULT_SORT_ORDER;
    }
    @Override
    public Uri getContentUri() {
    return CONTENT_URI;
    }
    @Override
    public String getContentType() {
    return CONTENT_TYPE;
    }
    @Override
    public String getContentItemType() {
    return CONTENT_ITEM_TYPE;
    }
    @Override
    public String[] getProjection() {
    return new String[] { Columns.NOTIZ_BETREFF, Columns.NOTIZ_TEXT, Co➥
lumns.NOTIZ_DATUM, Columns.NOTIZ_CONTACT_ID };
    }
    @Override
    public void createTable(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE " + Notizen.TABLE_NAME + " ("
                + Notizen.Columns._ID + " INTEGER PRIMARY KEY,"
                + Notizen.Columns.NOTIZ_TEXT + " TEXT,"
                + Notizen.Columns.NOTIZ_BETREFF + " TEXT,"
                + Notizen.Columns.NOTIZ_DATUM + " INTEGER,"
                + Notizen.Columns.NOTIZ_CONTACT_ID + " INTEGER"
                + ");");
    }

    public void upgradeTable(SQLiteDatabase db, int oldVersion, int newVer➥
    sion)
```

```
    {

    }

    public Uri insert(SQLiteDatabase db, ContentValues values)
    {
    if (!values.containsKey(Columns.NOTIZ_DATUM))
    {
    Long now = Long.valueOf(System.currentTimeMillis());
            values.put(Notizen.Columns.NOTIZ_DATUM,now);
    }
    return ContentUris.withAppendedId(getContentUri(),
db.insert(getTableName(), Columns.NOTIZ_BETREFF, values));
    }

    public int update(SQLiteDatabase db, ContentValues values, String where,
String[] whereArgs, String id)
    {
    String _where = DatabaseProvider.createWhere(where,id);
    return db.update(getTableName(),values,_where,whereArgs);
    }

    public int delete(SQLiteDatabase db, String where, String[] whereArgs,
String id)
    {
    String _where = DatabaseProvider.createWhere(where,id);
    return db.delete(getTableName(),_where,whereArgs);
    }
}
```

Dieser Ansatz sieht ja schon sehr viel übersichtlicher aus. Wenn wir nun eine weitere Tabelle hinzubekommen, können wir uns an diesem Schema orientieren, ohne dass es zu wahnsinnig aufgeblähten `switch(…)`-Statements zum Unterscheiden der einzelnen Tabellen führt.

Unseren Content-Provider können wir nun verwenden. Im Grunde geht das genauso, wie wir das beim MediaStore oder den Contacts gemacht haben.

Listing 3.150: **Anlegen einer neuen Notiz**

```
private void neueNotiz() {
    ContentValues values = new ContentValues();
    values.put(Notizen.Columns.NOTIZ_BETREFF,getResources().getString(R.
string.neue_notiz));
    Uri uri = getActivity().getContentResolver().insert(Notizen.CONTENT_URI,
values);
    startActivity(new Intent(Intent.ACTION_EDIT,uri));
}
```

Im obigen Listing wird ein neuer Eintrag in unserem Content-Provider angelegt. Hier sehen wir sehr schön, dass wir die gleichen Mechanismen nutzen wie bei »fremden« Content-Providern.

Das Ändern und Aktualisieren von Notizen finden wir in der EditNotizActivity, die an das Bearbeiten der MediaStore-Metadaten angelehnt ist und genauso funktioniert, mit dem Unterschied, dass wir nun unseren ContentProvider ansprechen.

Listing 3.151: **Aktualisieren einer Notiz**

```
public void mapDataFromView()
{
    Intent intent = getIntent();
    […]
    Uri data = intent.getData();
    […]
    ContentValues values = new ContentValues();
    EditText title = (EditText)dialogView.findViewById(R.id.notiz_betreff);
    EditText description = (EditText)dialogView.findViewById(R.id.notiz_
text);
    if (title!=null) values.put(Notizen.Columns.NOTIZ_BETREFF, title.get➥
    Text().toString());
    if (description!=null) values.put(Notizen.Columns.NOTIZ_TEXT, descrip➥
    tion.getText().toString());
    getContentResolver().update(data, values, null, null);
    […]
}
```

TIPP

Natürlich muss man sich im Zusammenhang mit der SQLite Datenbank auch mit SQL beschäftigen. Ein Einstieg, mitunter sehr technisch, da es auch auf das Einbetten der Datenbank eingeht, ist `http://www.sqlite.org/`**. Einen Überblick kann man sich zudem unter** `http://de.wikipedia.org/wiki/SQL` **verschaffen.**

3.15 Loader

Android 3 stellt mit dem Loader-Konzept für Activities und Fragmente einen schlanken und effektiven Mechanismus bereit, benötigte Daten asynchron im Hintergrund zu laden. Darüber hinaus stellt das Loader-Framework sicher, dass Änderungen an den angefragten Datenquellen automatisch neu geladen werden. Außerdem sorgen die Loader dafür, dass nach einer Unterbrechung des Ladevorgangs wieder an der richtigen Stelle aufgesetzt wird, ohne dass alle Daten erneut gelesen werden.

Offensichtlich ist dieses Framework für Datenbankabfragen innerhalb unserer Applikation zu nutzen, aber auch um Daten oder Streams aus dem Netz zu laden oder auch um z.B. im Hintergrund Vorschaubilder zu berechnen.

Kern ist der LoaderManager, den wir mittels getLoaderManager() auf einer Activity oder einem Fragment anfordern können und der dazu dient, den eigentlichen Ladeprozess anzustoßen und zu kontrollieren. Mittels des Managers können beliebig viele Ladeprozesse gestartet werden, die über die LoaderCallbacks mit unserer Anwendung kommunizieren.

Das Framework stellt zwei Implementierungen des Loaders bereit, den AsyncTaskLoader und den CursorLoader. Der CursorLoader selbst ist eine Spezialisierung des AsyncTask-Loaders für Abfrage von Content-Providern. Den AsyncTaskLoader können wir für beliebige Daten benutzen.

TIPP

Die Loader hängen unmittelbar mit dem Lebenszyklus der Activity zusammen. Wenn die Activity schlafen geht, dann werden auch die Loader schlafen gelegt und wieder aufgeweckt, wenn die Activity weiterläuft. Um lange Vorgänge wie Synchronisierungen oder größere Downloads zu verwalten, bieten sich Services bzw. die Implementierung von Sync-Adaptern an, die diese Aufgaben im Hintergrund erledigt.

Im einfachsten Fall benutzen wir den Loader-Service innerhalb einer Activity bzw. eines Fragments, die/das auch die Schnittstellen für die Loader-Callbacks implementiert:

Listing 3.152: **Einfache Anwendung des Cursor-Loaders**

```
public class ShowImagesFragment extends ListFragment implements LoaderMana➥
    ger.LoaderCallbacks<Cursor> {
[…]
@Override
public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    contentUri = MediaStore.Images.Media.EXTERNAL_CONTENT_URI;
[…]
    cursorAdapter = new SimpleCursorAdapter(getActivity(),
    R.layout.imageview_listitem,
    null,
    DISPLAY ,
    new int[] {R.id.imageview_title,R.id.imageview_description,R.
id.imageview_image});
    setListAdapter(cursorAdapter);
    getLoaderManager().initLoader(0, null, this);
}
[…]
@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    return new CursorLoader(getActivity(),contentUri, PROJECTION, "", null,
MediaStore.Images.ImageColumns.DATE_TAKEN+" desc");
}
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    cursorAdapter.swapCursor(data);
}
@Override
public void onLoaderReset(Loader<Cursor> loader) {
    cursorAdapter.swapCursor(null);
}
[…]
}
```

Wichtig sind die Callbacks in denen der Loader erzeugt und der geladene Cursor an den Cursor-Adapter gebunden wird, wenn das Laden fertig ist.

Der Aufruf von `getLoaderManager().initLoader(int id, Bundle arguments, LoaderManager.LoaderCallBacks<Cursor> loaderCallbacks)` kann durch die Übergabe unterschiedlicher IDs dazu benutzt werden, mehrere Loader parallel zu starten. Innerhalb des Callbacks `onCreateLoader(int id, Bundle args)` erzeugen wir je nach ID den entsprechenden Loader, und über das Bundle lassen sich Parameter an die Loader übergeben.

Im Callback `onLoadFinished(Loader<Cursor> loader, Cursor data)` müssen wir dann die notwendigen Aktionen ausführen um die geladenen Daten anzuzeigen oder anderweitig zu verarbeiten. In unserem Beispiel wird der Cursor einfach an den Cursor-Adapter gebunden, und damit wird dann auch die Anzeige (hier: die ListView) aktualisiert. Der Loader kümmert sich selbst um den Lebenszyklus des Cursors, wir müssen den Cursor also niemals selbst schließen.

Der Callback `onLoadReset(Loader<Cursor> loader)` wird aufgerufen, wenn der Loader zurückgesetzt und damit die Daten, die durch den Loader verwaltet werden, nicht mehr verfügbar sind. Hier müssen wir die notwendigen Aktionen ausführen, um alle Referenzen auf die Daten zu lösen. Hier binden wir einfach einen null-Wert an den Cursouradapter, so dass der Adapter keine Datenquelle mehr hat, die er benutzen kann.

Mit dem Loader-Framework lässt sich unsere Anwendung auch leicht durchsuchbar machen. Die Methode `getLoaderManager().restartLoader(int id, Bundle args, LoaderManager.LoaderCallBacks<Cursor> loaderCallbacks)` dient dazu, den Loader neu zu starten und die aktuellen Daten zu holen. Im `onCreateLoader(…)`-Callback können wir dann z.B. die SQL-Abfrage so gestalten, dass der aktuelle Suchtext, der über ein Such-Widget in der Action Bar eingegeben wurde, berücksichtigt wird:

Listing 3.153: **Reagieren auf die Eingabe von Suchtext**

```
@Override
public boolean onQueryTextChange(String newText) {
    currentFilter = !TextUtils.isEmpty(newText) ? newText : null;
    getLoaderManager().restartLoader(0, null, this);
    return true;
}
[…]
```

Durch `restartLoader(…)` wird der Loader mit der ID 0 neu erzeugt, die aktuellen Daten werden abgeholt und, sobald die Daten vorliegen, wieder `onLoaderFinished(…)` aufgerufen.

Innerhalb von `onCreateLoader(…)` müssen wir dann noch den aktuellen Suchfilter berücksichtigen:

Listing 3.154: **Berücksichtigen des Suchtextes bei der Erzeugung des Loaders**

```
@Override
public Loader<Cursor> onCreateLoader(int arg0, Bundle arg1) {
    String selection = "";
    String[] selectionArgs = null;
    if (currentFilter!=null)
    {
```

```
        selection = "("+Notizen.Columns.NOTIZ_BETREFF+ " like ?)";
        selectionArgs = new String[] { "%"+currentFilter+"%" };
    }
    return new CursorLoader(getActivity(),contentUri, PROJECTION, selection,
selectionArgs, Notizen.DEFAULT_SORT_ORDER);
}
```

Wie hier schön zu sehen ist, lassen sich mittels des Loader-Frameworks, speziell bei der Verwendung des Cursor-Loaders, sehr einfach Anwendungen bauen, die geschmeidig auf Daten zugreifen können. Ein weiterer Nebeneffekt des Loader-Frameworks ist, dass Änderungen an den zugrundeliegenden Daten ebenfalls im Manager überwacht werden bzw. auf die Änderungen reagiert wird und wir uns um die Aktualisierung unserer Datenansichten hier nicht mehr kümmern müssen.

Eine allgemeinere Form der Loader sind die `AsyncTaskLoader`. Diese Loader starten einfach einen Hintergrund-Task, mit dem der Ladeprozess abgewickelt wird, und sind nicht wie die Cursor-Loader auf einen speziellen Mechanismus angewiesen. Wir können diese z.B. nutzen um Daten über das Internet zu ziehen, z.B. RSS-Feeds oder Streams von sozialen Netzwerken, neue Spielelevels oder Ähnliches.

Listing 3.155: **Laden der Level-Hintergründe in MarbleGameLibrary3**

```
public class LevelLoader extends AsyncTaskLoader<Level> {
Uri baseURI = Uri.parse("http://www.androidpraxis.de/downloads");
String foreground;
String middle;
String background;
NetworkUtilityMessageHandler networkMessageHandler;
public LevelLoader(Context context, NetworkUtilityMessageHandler network➥
    MessageHandler, String foreground, String middle, String background) {
    super(context);
    this.foreground = foreground;
    this.middle = middle;
    this.background = background;
    this.networkMessageHandler = networkMessageHandler;
}

@Override
public Level loadInBackground() {
    Level level = new Level();
    NetworkUtility nwu = new NetworkUtility(this.networkMessageHandler);
    level.foreground = nwu.loadBitmap(Uri.withAppendedPath(baseURI, this.
foreground));
    level.middle = nwu.loadBitmap(Uri.withAppendedPath(baseURI, this.
middle));
    level.background = nwu.loadBitmap(Uri.withAppendedPath(baseURI, this.
background));
    return level;
}

[…]
@Override
protected void onStartLoading() {
```

```
    super.onStartLoading();
    forceLoad();
}
}
```

Im Prinzip ist ein eigener Loader sehr einfach zu realisieren, in dem wir die Methode `loadInBackground(…)` überschreiben und eine Klasse schreiben, mit der wir die Ladeergebnisse zurückliefern können. Hier ist das die Klasse `Level`, mit der wir lediglich die drei Bitmaps transportieren.

Wir müssen dann noch die Methode `onStartLoading()` überschreiben, um mittels `forceLoad()` dafür zu sorgen, dass der Ladeprozess startet.

Benutzt wird der Loader dann folgendermaßen:

```
public class Game3 extends Game implements LoaderManager.
LoaderCallbacks<Level>, NetworkUtilityMessageHandler{
ProgressDialog progressDialog = null;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getLoaderManager().initLoader(0,null,this);
}
@Override
public Loader<Level> onCreateLoader(int id, Bundle args) {
    progressDialog = new ProgressDialog(this);
    progressDialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);
    progressDialog.setMessage("Lade Level...");
    progressDialog.setCancelable(false);
    progressDialog.show();
    return new LevelLoader(this,this,"hintergrund.png","mittelteil.
png","vordergrund.png");
}
```

Hier erzeugen wir unseren LevelLoader und übergeben die Activity auch als Handler für Fehlermeldungen.

```
@Override
public void onLoadFinished(Loader<Level> loader, Level level) {
    progressDialog.dismiss();
    getLoaderManager().destroyLoader(0);
}
```

An dieser Stelle ist der Loader fertig. Im Gegensatz zum Cursor-Loader zerstören wir unseren Loader zu diesem Zeitpunkt, um zu verhindern, dass die Bitmaps jedesmal geladen werden wenn die Activity lediglich aufgeweckt wird.

```
@Override
public void onLoaderReset(Loader<Level> loader) {
}

@Override
public void onException(final Throwable exception) {
    runOnUiThread(
    new Runnable()
    {
    public void run()
    {
    Toast.makeText(Game3.this, exception.toString(), Toast.LENGTH_SHORT).
show();
    }
    }
    );
}

@Override
public void onError(final StatusLine statusLine) {
    runOnUiThread(
    new Runnable()
    {
    public void run()
    {
    Toast.makeText(Game3.this, statusLine.getStatusCode()+" - "+statusLine.
getReasonPhrase(), Toast.LENGTH_SHORT).show();
    }
    }
    );
}
```

Die Methoden `onError` und `onException` werden aus dem Loader bzw. den Hilfsbibliotheken aus aufgerufen. Da der Loader aber in einem separaten Task läuft, dürfen wir den Toast nicht direkt aussprechen. Wir legen ihn durch den Aufruf von `runOnUiThread(…)` quasi auf Halde, der Code wird ausgeführt, sobald der UI-Thread wieder an die Reihe kommt. So können wir sicher Dinge zwischen Hintergrund-Threads und dem UI-Thread vermitteln.

Listing 3.156: **Benutzen des LevelLoader**

```
}
```

Das Laden der Hintergrunddateien vom Server wird mittels des Apache-Http-Clients durchgeführt, mit dem wir sehr einfach Zugriffe auf Web-Server per Http-Protokoll realisieren können. Wir müssen unserer Anwendung dafür die Erlaubnis `android.permission.INTERNET` zuweisen.

Listing 3.157: **Laden einer Bitmap-Ressource aus dem Web**

```
public Bitmap loadBitmap(Uri uri)
{
    Bitmap result = null;
```

```
    HttpGet get = new HttpGet(uri.toString());
    try {
        HttpResponse response = httpClient().execute(get);
        if (response.getStatusLine().getStatusCode() == HttpStatus.SC_OK)
        {
            HttpEntity entity = response.getEntity();
            InputStream is = entity.getContent();
            result = BitmapFactory.decodeStream(is);
        }
        else
        {
            if (messageHandler!=null) messageHandler.onError(response.
getStatusLine());
        }
    } catch (ClientProtocolException e) {

    if (messageHandler!=null)
    {
        messageHandler.onException(e);
    }

    } catch (IOException e) {

        if (messageHandler!=null) messageHandler.onException(e);

}
return result;
}
```

INFO

Im MarbleGame3-Beispiel verwenden wir auch die Methode onRetainNonConfigurationInstance(), um den geladenen Level z.B. über das Drehen des Geräts hinaus zu retten.

Listing 3.158: **Ausnutzen von onRetainNonConfigurationInstance()**

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Object retained = getLastNonConfigurationInstance();
    if (retained == null || (!(retained instanceof Level)))
    {
        getLoaderManager().initLoader(0,null,this);
    }
    else
    {
        level = (Level)retained;
        setLevel(level);
    }
}
[…]
@Override
public Object onRetainNonConfigurationInstance() {
    return level;
}
```

3.16 Broadcast Receiver

Broadcast Receiver dienen dazu, Broadcast-Intent-Nachrichten zu empfangen. Das sind Intents, die nicht zur Ausführung einer Activity führen, sondern z.B. Informationen über Zustandsänderungen im Power-Management oder in anderen Subsystemen liefern, die in irgendeiner Weise von Interesse für andere Anwendungen sind.

Broadcast Receiver werden ähnlich wie Activites im Manifest deklariert oder innerhalb einer Anwendung mittels `registerReceiver(…)` bekannt gemacht.

Der Receiver reagiert auf den Broadcast mit `onReceive(…)` und sollte keine asynchronen Aktionen ausführen, d.h., alles, was in `onReceive(…)` passiert, muss auch dort enden. Nach dem `onReceive(…)` verlassen wurde kann der Receiver möglicherweise nicht mehr aktiv sein, es sei denn, es handelt sich um einen per `registerReceiver(…)` registriertes Objekt.

Aus dem Receiver heraus sollten demnach auch keine Dialoge erzeugt werden. Üblicherweise wird im Receiver eine Notification erzeugt, alternativ könnten wir einen Service oder auch eine Activity starten.

Im Zusammenspiel mit Services können wir bei Bedarf selbst Broadcasts versenden, z.B. wenn wir einen Service erstellen der irgendwelche Daten überwacht und bei einem bestimmten Signal einen entsprechenden Broadcast erzeugt, auf den wir oder andere dann reagieren können.

Im Abschnitt über Intents haben wir einige Broadcast-Actions kennengelernt. Bei der Anwendung ist noch darauf zu achten, dass manche Broadcasts nicht über eine Manifestdeklaration, sondern nur durch `registerReceiver(…)` behandelt werden können. So kann `Intent.ACTION_TIME_CHANGED`, das jede Minute ausgelöst wird, nur durch einen per `registerReceiver(…)` registrierten Broadcast-Receiver empfangen werden, `Intent.ACTION_POWER_CONNECTED` hingegen auch durch die Deklaration im Manifest.

Listing 3.159: **Deklaration des Receivers für das An- und Abstöpseln der Stromversorgung**

```
<receiver android:name="PowerBroadcastReceiver">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_POWER_CONNEC➥
    TED"></action>
    </intent-filter>
    <intent-filter>
    <action android:name="android.intent.action.ACTION_POWER_DISCONNEC➥
    TED"></action>
    </intent-filter>
</receiver>
```

Listing 3.160: **Implementierung des Receivers**

```
public class PowerBroadcastReceiver extends BroadcastReceiver {
@Override
public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, intent.getAction(), Toast.LENGTH_SHORT).show();
}
}
```

Alle Receiver, die wir im Manifest deklarieren können, können wir auch intern erzeugen und registrieren. Umgekehrt gilt das, wie gesagt, nicht.

Ein interessanter Broadcast könnte z.B. das Einstecken einer SD-Karte sein (`Intent.ACTION_MEDIA_MOUNTED`) oder aber eine Änderung in der Netzwerkkonnektivität (`ConnectivityManager.CONNECTIVITY_ACTION`).

Listing 3.161: **Empfangen von Netzwerkverbindungsnachrichten**

```
public class NetworkBroadcastReceiver extends BroadcastReceiver {

@Override
public void onReceive(Context context, Intent intent) {
    if (intent.getAction().equals(ConnectivityManager.CONNECTIVITY_ACTION))
    {
        NetworkInfo ni = (NetworkInfo)intent.getParcelableExtra(Connectivity
Manager.EXTRA_NETWORK_INFO);
        if (ni!=null)
        {
        String text = ni.getTypeName()+" "+ni.getSubtypeName();

        if (ni.isAvailable()) text+=" available";
        if (ni.isConnected()) text+=" connected";
        if (ni.isConnectedOrConnecting()) text+=" connecting...";
        if (ni.isFailover()) text+=" failover";
        if (ni.isRoaming()) text+=" roaming";
        Toast.makeText(context, text, Toast.LENGTH_SHORT).show();
        }
    }
}
}
```

Listing 3.162: **Registrieren des Receivers innerhalb einer Activity**

```
Intent-Filter Intent-Filter = new Intent-Filter(ConnectivityManager.CONNEC➥
    TIVITY_ACTION);
registerReceiver(new NetworkBroadcastReceiver(),Intent-Filter);
```

Welche Nachrichten zu verwenden sind und welche zusätzlichen Informationen die Intents transportieren, müssen wir in der Android-Dokumentation nachschlagen. Viele Broadcasts sind bereits in `Intent` deklariert, aber die verschiedenen Subsysteme deklarieren ihrerseits auch Broadcast-Actions, wie wir am Beispiel des `ConnectivityManagers` sehen.

Wie wir an den Beispielen sehen können, sind die Broadcast Receiver überall dort sinnvoll, wo wir auf die Änderung der »Umgebung« reagieren wollen. Manches wird durch das Ressourcensystem bereits abgedeckt, z.B. wenn wir ein alternatives Layout wählen wollen, wenn das Gerät in einem Dock platziert wird. Wenn wir darüber hinaus aber noch weitere Funktionen ausführen wollen, dann sind die Receiver das Mittel der Wahl.

3.17 Services

Services können Hintergrundaufgaben übernehmen, die keine Interaktion mit dem Benutzer erfordern und auch weiterlaufen sollen, wenn andere Anwendungen im Vordergrund laufen. Musik soll z.B. auch weiterlaufen, wenn wir gerade eine E-Mail verfassen oder auf Webseiten surfen. Darüber hinaus können Hintergrunddienste periodisch irgendetwas überwachen, z.B. die aktuelle Position, oder Statusmeldungen aus dem Netz empfangen und ggf. eine Benachrichtigung auslösen.

Das Herunterladen von Updates und auch die Synchronisierung von Diensten (Kontakte, Bilder etc.) finden auch mittels Services statt.

Grundsätzlich ist die Verwendung von Services mit Bedacht zu wählen, denn je nach Priorität und Ressourcenverbrauch erhöhen Hintergrunddienste den Stromverbrauch – ein häufiger Kritikpunkt am sehr mächtigen Multitasking-Konzept von Android.

Die einfachste Methode einen Service zu implementieren, ist, die Klasse `IntentService` zu benutzen. Diese Klasse liefert den Rahmen für Services, die nur **eine** Aufgabe ausführen und keine nebenläufigen Aufgaben erledigen sollen. Services, die mehrere Aufgaben abarbeiten, sind komplexer in der Implementierung und können bei unsachgemäßer Ausführung zu Lasten der Performance und des Energieverbrauchs gehen. Die meisten Hintergrundservices bearbeiten sowieso immer nur eine Aufgabe, z.B. in gewissen Abständen etwas zu prüfen und ggf. ein Signal zu schicken, wenn ein bestimmtes Ereignis eintritt, oder im Hintergrund etwas herunterzuladen.

Listing 3.163: **Einfacher Service**

```
public class SimulatedBackgroundService extends IntentService {

private static final int NOTIFICATION_ID = 1;

public SimulatedBackgroundService() {
      super("SpielwieseSimulatedBackgroundService");
  }

private void createStausBarNotification(CharSequence text)
{
    NotificationManager notificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);

    int icon = R.drawable.icon;
    CharSequence tickerText = text;
    long when = System.currentTimeMillis();
    Context context = this;
```

```
    CharSequence contentTitle = "Spielwiese Hintergrundservice";
    CharSequence contentText = text;

    Intent notificationIntent = new Intent(this, StartedFromNotificationAc➥
    tivity.class);
    PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notifi➥
    cationIntent, 0);

    Notification notification = new Notification.Builder(context)
      .setSmallIcon(icon)
      .setTicker(tickerText)
      .setContentTitle(contentTitle)
      .setContentText(contentText)
      .setWhen(when)
      .setContentIntent(contentIntent)
      .getNotification();

    notificationManager.notify(NOTIFICATION_ID, notification);
}
@Override
protected void onHandleIntent(Intent intent) {
    createStausBarNotification("SimulatedBackgroundService onHandleIntent
started");
    long endTime = System.currentTimeMillis() + 15*1000;
    while (System.currentTimeMillis() < endTime) {
    synchronized (this) {
        try {
        wait(endTime - System.currentTimeMillis());
        } catch (Exception e) {
        }
        }
    }
    createStausBarNotification("SimulatedBackgroundService onHandleIntent
finished");
}
@Override
public void onDestroy() {

Toast.makeText(this, "SimulatedBackgroundService onDestroy", Toast.LENGTH_
SHORT).show();
super.onDestroy();

}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
Toast.makeText(this, "SimulatedBackgroundService started", Toast.LENGTH_
SHORT).show();
return super.onStartCommand(intent, flags, startId);
}
}
```

Das obige Beispiel simuliert einen 15 Sekunden laufenden Service. Statt der Zeit-Schleife können hier die Aufgaben ausgeführt werden, die wir ausführen möchten.

Dieser Service kann dann mit

```
Intent intent = new Intent(this,SimulatedBackgroundService.class);
startService(intent);
```

oder

Listing 3.164: **Starten des Service**

```
Intent intent = new Intent(this,"de.androidpraxis.SpielwieseLibrary3.Simula➥
    tedBackgroundService");
startService(intent);
```

gestartet werden.

ACHTUNG

Damit eine andere Anwendung unseren Service starten kann – so wir das denn erlauben –, muss die andere Anwendung den Namen kennen, eine andere Möglichkeit gibt es nicht. Wir sollten also möglichst den Namen nicht mehr ändern, wenn wir unsere Anwendung einmal publiziert haben.

Die Klasse `IntentService` sorgt dafür, dass `onHandleIntent(…)` bereits in einem eigenen Thread abläuft, wir müssen also keine weiteren Threads abspalten. Wenn wir das selbst realisieren wollten, müssen wir direkt von `Service` ableiten und einiges mehr an Verwaltungsarbeit übernehmen. Dann könnten wir in unserem Service auch mehrere Startanforderungen auf parallele Threads aufteilen. Eine typische Anwendung dafür wäre ein Download-Manager, dem wir per Intent die zu ladende Ressource übergeben können und der für jeden Download einen eigenen Thread abspaltet.

INFO

Android bietet seit dem API-Level 9 einen solchen Download-Manager als Systemservice an, für diese Anwendung brauchen wir also keinen eigenen Service zu schreiben.

Grundsätzlich ließe sich ein solcher Service folgendermaßen realisieren:

```
public class SimulatedMultiThreadedBackgroundService extends Service {
private static final int NOTIFICATION_ID = 1;
static final public String EXTRA_SERVICE_ARGUMENT = "de.androidpraxis.Spiel➥
    wieseLibrary3.SimulatedMultiThreadedBackgroundService.SERVICE_ARGUMENT";

public SimulatedMultiThreadedBackgroundService() {
    super();
}
private void createStausBarNotification(int id, CharSequence text)
{
    […]
    notificationManager.notify(id, notification);
}
protected void doWork(int id, Bundle args)
{
    long msecs = (long)(60+(Math.random()*60))*1000;
    long endTime = (System.currentTimeMillis() + msecs);
```

```
    createStausBarNotification(id, args.getString(EXTRA_SERVICE_ARGUMENT)+"
started "+msecs/1000+"s");
    while (System.currentTimeMillis() < endTime) {
        synchronized (this) {
        try {
        wait(endTime - System.currentTimeMillis());
        } catch (Exception e) {
        }
        }
    }
    createStausBarNotification(id, args.getString(EXTRA_SERVICE_ARGUMENT)+"
finished");
}

private Looper mServiceLooper;
private ServiceHandler mServiceHandler;

private final class ServiceHandler extends Handler {
    public ServiceHandler(Looper looper) {
        super(looper);
    }
    @Override
    public void handleMessage(Message msg) {
        doWork(msg.arg1,msg.getData());
        stopSelf(msg.arg1);
    }
}
```

Der `ServiceHandler` dient dazu Nachrichten aus einer Warteschlange zu empfangen, wir senden später eine Nachricht an diesen Handler, um die Arbeit des Service durchzuführen. Der Handler wird in der nächsten Methode an einen `HandlerThread` gebunden. Ein HandlerThread ist wie ein herkömmlicher Thread, mit dem Unterschied, dass er eine Nachrichtenwarteschlange (einen `Looper`) bereitstellt, über die Nachrichten an den Thread und damit an die angehängten Handler geschickt werden können.

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
        Toast.makeText(this, "SimulatedMultiThreadedBackgroundService
started #"+startId, Toast.LENGTH_SHORT).show();
        String argument = intent.getStringExtra(EXTRA_SERVICE_ARGUMENT);

        HandlerThread thread = new HandlerThread("SimulatedMultiThreadedBack
groundService",
        Process.THREAD_PRIORITY_BACKGROUND);

        thread.start();

        mServiceLooper = thread.getLooper();
        mServiceHandler = new ServiceHandler(mServiceLooper);

        Message msg = mServiceHandler.obtainMessage();
        msg.arg1 = startId;
        Bundle args = new Bundle();
        args.putString(EXTRA_SERVICE_ARGUMENT, argument);
```

```
        msg.setData(args);

        mServiceHandler.sendMessage(msg);
        return Service.START_NOT_STICKY;
}
```

In `onStartCommand(…)` wird nun aus dem Intent das Argument extrahiert, das an den Worker-Thread übergeben wird. In einem »echten« Service könnte das z.B. die Adresse einer Datei zum Download sein. Dann werden der `HandlerThread` und der `ServiceHandler` erzeugt, an den dann eine entsprechende Nachricht geschickt wird, die die ID der Anforderung und ein `Bundle` mit dem Service-Argument enthält.

Listing 3.165: **Service mit parallelen Threads**

```
@Override
public void onDestroy() {
    Toast.makeText(this, "SimulatedMultiThreadedBackgroundService onDest-
roy", Toast.LENGTH_SHORT).show();
    super.onDestroy();
}
@Override
public IBinder onBind(Intent arg0) {
    return null;
}
}
```

Handler-Threads bzw. Handler lassen sich auch für Multithreading und andere Aufgaben in Activities oder Fragmenten nutzen. Der Vorteil ist, dass man über die Handler mittels Nachrichten kommunizieren kann, z.B. um die Verarbeitung zu steuern oder aus einem Thread Nachrichten an den UI-Thread einer View oder Activity zu schicken.

Starten lassen sich diese Services genauso wie der IntentService, mit dem Unterschied, dass wir dem Intent noch ein Argument mitgeben:

Listing 3.166: **Starten mehrerer Anforderungen an den Service**

```
case MENU_START_MULTITHREADED_SERVICE:
for (int i=1; i<4; i++)
{
    Intent intent1 = new Intent(this,SimulatedMultiThreadedBackgroundServi
ce.class);
    intent1.putExtra(SimulatedMultiThreadedBackgroundService.EXTRA_SERVICE_
ARGUMENT, "Thread #"+i);
    startService(intent1);
}
break;
```

Die Königsdisziplin sind sogenannte gebundene Services (Bound Service), bei denen der Service nicht per Intent gestartet wird, sondern eine Bindung mit dem Service angefordert wird. So lange Bindungen an den Service existieren, »lebt« der Service. Über die Bindung kann über eine definierte Schnittstelle mit dem Service kommuniziert werden, um bestimmte Methoden im Service aufzurufen. Diese Art von Service wird dann benutzt, wenn

komplexere Anforderungen und Steuerungsaufgaben oder aber auch eine über die Kommunikation mit Intents hinausgehende prozessübergreifende Kommunikation nötig ist.

Beispiele für die Nutzung von Bound Services sind:

1. Application Widgets
2. Authentication Services
3. Sync-Adapter

Im Falle von Application Widgets kommunizieren sogenannte Remote Widgets über eine Bindung mit der eigentlichen Anwendung, ohne dass eine Activity im Vordergrund stünde. App-Widgets sind sozusagen eingebettete aktive Widgets in anderen Anwendungen, z.B. der Homescreen.

Authentication Services werden für Synchronisierungsaufgaben verwendet und benutzen die im System konfigurierten Accounts, um auf Ressourcen wie ein Google-Konto, ein Facebook-Konto oder andere Konten transparent zuzugreifen.

Der Sync-Adapter wiederum führt den eigentlichen Abgleich mit dem Konto durch.

Diese Funktionalitäten werden als Bound Services realisiert.

3.18 Zwischenbilanz

Das war nun die sogenannte Pflicht. Wir haben die Grundlagen des Android-Systems kennengelernt und die wichtigen Bausteine einer jeden Anwendung. Wir können mit Ressourcen umgehen, Layouts erstellen und Anwendungen aus Activities, Fragmenten und Views zusammenbauen. Wir können auf Ereignisse in den Views reagieren, Menüs erstellen und die Action Bar nutzen. Darüber hinaus haben wir Content-Provider angezapft und eigene Provider erstellt sowie Services und Broadcast Receiver kennengelernt.

All das ist eine gute Grundlage für das, was nun folgt: Die Kür.

Im nächsten Kapitel beschäftigen wir uns mit all den Dingen, die in den Anwendungen Spaß machen:

1. App-Widgets
2. Grafik
3. Sensoren
4. Location Services
5. Die Kamera
6. Audio und Video
7. Near Field Communication
8. u.a.m.

Also, schnallen wir uns an und begeben uns auf die weitere Reise.

4 Die Tiefen von Android



Alles, was wir im vorigen Kapitel besprochen haben, waren die notwendigen Grundlagen, um Anwendungen für Android zu entwickeln. Einige weiterführende Themen haben wir dabei schon angekratzt, wie die Ausgabe von Grafik, andere Themen wie das Sensormanagement, die Kamerahardware und Multimedia haben wir noch gar nicht betrachtet, die die Geräte aber doch so spannend machen.

Dafür wollen wir nun ein wenig in die Tiefen gehen.

4.1 Grafik

Wenn wir Layouts für unsere Anwendungen erstellen und Widgets benutzen, haben wir schon ein wenig über Grafik in Android erfahren. Auch die Beschäftigung mit dem Ressourcensystem hat uns in die Nähe der 2D-Grafik geführt. Im Abschnitt über das Erstellen eigener Widgets sind wir dem Ganzen noch viel näher gekommen und haben die Methode `onDraw(Canvas canvas)` der Basisklasse `View` kennengelernt, und hier kommen wir direkt an den Punkt, wo wir selbst mit Grafik arbeiten können: der Canvas, die Leinwand, auf der das Android-System Widgets, Bilder und vieles mehr zeichnet und auf der wir selber malen können.

Im Bereich der 2D-Grafik gibt es zwei Methoden, die im Grunde beide wieder bei der Leinwand landen, sich aber im Detail in der Abarbeitung der »Malaufträge« und mithin in der Performance unterscheiden:

1. Das Zeichnen auf dem Canvas in onDraw(...)
2. Das direkte Zeichnen auf den Bildschirm mittels SurfaceHolder

Die erste Option ist einfach zu realisieren und bestens geeignet für die Gestaltung eigener Widgets und für einfache grafische Darstellungen inklusive einfacher Animation. Da der Aktualisierungsprozess innerhalb des View-Systems stattfindet und innerhalb des UI-Threads, ist der limitierende Faktor die Performance des UI-Threads. Damit ist dieses Verfahren **nicht** für komplexe, zeitkritische Animationen geeignet.

Die zweite Option ist komplexer, bietet aber die Möglichkeit, Grafiken innerhalb eines eigenen Threads und damit auch in höherem Aktualisierungsintervall zu zeichnen als innerhalb

der Views. Die `Surface` ist auch ein ziemlich direkter Zugang zum Framebuffer (dem Bildschirmspeicher) und damit hardwarenäher als das Zeichnen innerhalb des View-Systems. Der `SurfaceHolder` stanzt, bildlich gesprochen, ein Loch in die View und gibt den fast unmittelbaren Zugriff auf den Framebuffer frei. Dieses Verfahren ist also für Spiele, komplexe Animationen, Video-Playbacks und z.B. die Kameravorschau mit ihrem kontinuierlichen Datenstrom geeignet.

4.1.1 Zeichnen in View.onDraw(...)

Im Abschnitt über das Erstellen eigener Widgets durch Ableiten von der Basisklasse `View` haben wir unter anderem die `onDraw(…)`-Methode kennengelernt. Diese Methode wird vom View-System aufgerufen, wenn die View neu gezeichnet werden muss. Das ist z.B. dann der Fall, wenn das Widget zum ersten Mal dargestellt wird, das Layout sich verändert oder ein darüber liegendes Fenster wieder verschwindet.

All diese Ereignisse werden durch das System selbst überwacht und entsprechend umgesetzt, wir müssen einfach die `onDraw(…)`-Methode mit unseren Befehlen füllen.

Wenn wir allerdings Animationen einsetzen oder auf Wertänderung von Sensoren reagieren wollen, dann müssen wir für die Aktualisierung zum richtigen Zeitpunkt selber sorgen. Um ein Widget neu zu zeichnen, nutzen wir die Methode `View.invalidate()` bzw. `View.postInvalidate()`.

ACHTUNG

Die `onDraw(…)`-Methode muss immer im UI-Thread ausgeführt werden. Darauf müssen wir besonders dann achten, wenn wir mehrere Threads verwenden, z.B. um periodische Aktualisierungen oder andere Hintergrundaufgaben auszuführen. Wenn wir die Aktualisierung aus einem anderen Thread auslösen wollen, dann müssen wir `View.postInvalidate()` benutzen. Diese Methode legt den Aufruf in der Warteschlange des UI-Threads ab, sodass der Aufruf sicher im UI-Thread erfolgt.

Listing 4.1: **Überschreiben der Methode onDraw(...)**

```
public class CompassView extends View implements SensorReceiver {
[…]
    @Override
    public void setValues(float[] values) {
        lastKnownValues = values.clone();
        postInvalidate();
    }
    […]
    @Override
    protected void onDraw(Canvas canvas)
    {
        […]
    }
[…]
}
```

Im obigen Beispiel kann die `CompassView` Wertänderungen von Sensoren empfangen, es wird davon ausgegangen, dass hier von irgendwoher die Richtungsangabe in Grad einläuft. Immer dann, wenn sich dieser Wert ändert, wird `postInvalidate(…)` aufgerufen. Damit stellen wir sicher, dass die Aktualisierung auch dann funktioniert, wenn die Werte durch einen anderen Thread geliefert werden.

Die eigentlichen Zeichenoperationen finden dann in `onDraw(…)` auf dem übergebenen Canvas statt. Den Canvas betrachten wir im Folgenden genauer.

4.1.2 Der Canvas und das Paint-Objekt

Der Canvas (Leinwand) ist ein Objekt, auf dem Grafikoperationen ausgeführt werden können. Der Canvas stellt Methoden bereit, um grafische Primitiven wie Linien, Ellipsen, Rechtecke zu zeichnen und komplexere Rasteroperationen wie die Darstellung von Bitmap-Grafiken oder Fülloperationen durchzuführen.

Der Canvas selbst ist dabei ein abstraktes Modell, das wiederum unterschiedliche Ausgabeziele für die Zeichenoperationen ansprechen kann. Im View-System wird das in der Regel so realisiert, dass zu Beginn einer Bildschirmaktualisierung ein entsprechend großer Puffer (Framebuffer bzw. eine Bitmap) erstellt wird, auf den die Zeichenoperationen der einzelnen (sichtbaren) Views angewendet werden. Nachdem alle Views ihre Zeichenprogramme abgespult haben, wird der Puffer an den Bildschirm geschickt, et voilà, die Aktualisierungen werden sichtbar.

Wir können einen Canvas aber auch selbst mit einer eigenen Bitmap verbinden und darauf dann Zeichenoperationen ablaufen lassen. Diese Bitmap können wir dann speichern, übertragen oder irgendwo anders darstellen.

INFO

In früheren Bildschirmsystemen war es oft so, dass die Ausgabe direkt auf den Bildschirmspeicher erfolgte und durch die Latenz des Bildschirmaufbaus ein Flackern bei schnellen Aktualisierungen entstand. Um das zu vermeiden, hat man den oben beschriebenen Weg benutzt und das Bild erst einmal komplett in einem unabhängigen Speicher aufgebaut und dann den Speicherinhalt in einem Rutsch an den Bildspeicher geschickt. Diese Technik nennt sich Doppelpufferung (Double Buffering). Moderne Systeme führen dieses in der Regel bereits im Betriebssystem oder im Falle von hardwarebeschleunigten Grafikchips direkt auf der Grafikhardware aus.

Ein weiteres Ausgabeziel, auf das mit einem Canvas zugegriffen werden kann, ist das `Picture`. Ein `Picture` zeichnet alle Zeichenoperationen als Befehle auf und kann diese auf einem anderen Canvas abspielen. Das `Picture` verwaltet die Zeichenbefehle aber nicht als Befehle des Frameworks, sondern in einer Form, die dem Grafiktreiber sehr nahe steht. Dadurch können die Zeichenbefehle schneller ausgeführt werden als die äquivalenten Java-Aufrufe. Damit bietet sich das `Picture` ggf. als Zwischenspeicher für komplexere Zeichnungen an, z.B. in einer Zeichenanwendung, um die Darstellung zu optimieren.

Da das Picture auch in einem Stream, somit in einer Datei, gespeichert werden kann, können Bilder für eine spätere Verwendung zwischengespeichert werden.

Diese Methode bietet sich an, wenn das Bild aus Grafikprimitiven aufgebaut wird, da der Speicherverbrauch niedriger ist als beim Zeichnen auf eine entsprechende Bitmap. Sobald aber Rastergrafiken (Bitmaps) ins Spiel kommen, muss man abwägen, ob ein `Picture` oder eine Bitmap als Zwischenspeicher infrage kommt.

Wenn wir den direkten Zugriff per `SurfaceHolder` wählen, benutzen wir einen Canvas, der mit einer `Surface` verbunden ist.

Egal worauf der Canvas aber letztendlich zeichnet, die Operationen, die wir zur Verfügung haben, sind immer die gleichen.

Neben den Grafikprimitiven und komplexeren Operationen besitzt der Canvas ein Koordinatensystem und eine Transformationsmatrix. Mittels der Transformationsmatrix können wir das Koordinatensystem beliebig transformieren, um weitere Effekte zu erreichen:

1. Rotation
2. Skalierung
3. Translation (Verschiebung)
4. Scherung

Die Anwendung der Matrix findet während der Zeichenoperationen auf alle zu zeichnenden Punkte statt. Gerade in Verbindung mit Drawables oder in Verbindung mit Pfaden (Path) ergeben sich vielfältige Gestaltungsmöglichkeiten.

Eine Kompassnadel können wir z.B. als Pfadobjekt erstellen, die durch den Sensor übergebene Himmelsrichtung wird einfach vor dem Zeichnen der Nadel als Rotation festgelegt. Haben wir hingegen ein schönes PNG-Bild einer Kompassrose, benutzen wir dieses als Drawable und rotieren den Canvas vor dem Zeichnen des Drawables.

Neben der reinen 2D-Transformationsamtrix ist die Klasse `android.graphics.Camera` sehr interessant. Diese Klasse repräsentiert **nicht** die Kamera zum Aufnehmen von Bildern oder Videos, sondern eine virtuelle Kamera in unserem Koordinatensystem. Der Begriff Kamera wird in der 3D-Technik als Synonym für den Beobachterstandpunkt benutzt, die Kamera »filmt« sozusagen die 3D-Szene, und heraus kommt ein 2D-Bild, das der Projektion der 3D-Szene auf den Film entspricht. Mit der Kamera können wir Rotationen und Translationen im Raum um alle drei Achsen (X-Achse, Y-Achse und Z-Achse) beschreiben sowie den Beobachtungspunkt der Kamera im Raum festlegen. Aus diesen Angaben errechnet sich eine Matrix, die wiederum auf einen Canvas oder auf View-Animationen angewendet werden kann. Damit ist es möglich, dreidimensionale Effekte im zweidimensionalen Raum des Canvas oder der Views zu simulieren.

Das nahezu wichtigste Objekt im Zusammenhang mit der 2D-Grafik ist das `Paint`-Objekt, mit dem nahezu alle Zeichenoperationen parametrisiert werden können.

Das `Paint`-Objekt transportiert Stil- und Farbinformationen für die Grafikoperationen. Mit dem Paint-Objekt können wir Farbe und Textstile setzen, aber auch Filtereffekte definieren, den Alpha-Kanal (Transparenzkanal) festlegen und den Stil des Zeichenstifts bezüglich Dicke, Linienenden und Linienverbindungen angeben.

Mit dem `Paint`-Objekt können auch Shader sowie ein Shadow-Layer definiert werden. `Shader` werden benutzt, um Farbverläufe zu erstellen oder Texturen zu erzeugen. Der `Shader` bestimmt über seinen Algorithmus das Aussehen jedes zu zeichnenden Pixels. Der Shadow-Layer dient dazu, alle Zeichenoperationen mit einem Schatten zu versehen.

`Shader` können nicht dazu benutzt werden, um Bitmaps zu manipulieren. Dazu können aber `ColorFilter` eingesetzt werden. Ein `ColorFilter` manipuliert auf Basis eines Algorithmus die Farbe jedes zu zeichnenden Bildpunkts.

Die wichtigsten Methoden des Paint-Objekts sind:

setColor(int color) setARGB(int a, int r, int g, int b) setAlpha(int a)	Setzen der Farbe. Alpha ist der Wert der Transparenz, von 0 (entspricht durchsichtig) bis 255 (entspricht vollständig deckend). Auf Bildschirmen wird ausschließlich das RGB-Modell (inklusive Alpha-Kanal) verwendet, das die Farben aus der additiven Mischung der Werte für Rot, Grün und Blau darstellt. Die Hilfsklasse Color besitzt einige Methoden, um RGB-Werte aus dem HSV-Modell umzurechnen. HSV bedeutet Hue, Saturation und Value und beschreibt die Farben durch die Lage in einem Farbkreis, die Sättigung und die Helligkeit. **Achtung:** Es gibt keine separaten Farben für den Stift und die Füllung der Figuren. Wenn wir also ein ausgefülltes Rechteck und die Kontur des Rechtecks in einer unterschiedlichen Farbe zeichnen wollen, dann müssen wir die Figur zweimal zeichnen, einmal mit dem Stil Paint.Style.FILL und der Füllfarbe und dann mit Paint.Style.STROKE und der Konturfarbe.
setStyle(Paint.Style style)	Setzt den Zeichenstil. Hier können wir angeben, ob unsere Figuren und Pfade mit Umrissen, mit Umrissen und ausgefüllt oder nur ausgefüllt gezeichnet werden sollen: Paint.Style.FILL: Nur Füllen (der Umriss wird nicht gezeichnet) Paint.Style.FILL_AND_STROKE: Ausfüllen und Umriss zeichnen Paint.Style.STROKE: Es wird nur der Umriss gezeichnet.

Tabelle 4.1: **Wichtigste Methoden des Paint-Objekts**

setStrokeCap(Paint.Cap cap)	Mit Strokes (Striche) sind alle Linienzüge gemeint, die zum Zusammensetzen der grafischen Figuren benutzt werden, also wenn Linien gezogen, Pfade oder geometrische Primitiven gezeichnet werden. Cap beschreibt hier den Anfang und das Ende eines Strichs. Paint.Cap.BUTT: glatter Anfang und glattes Ende, Abschluss direkt am Ende bzw. am Anfang der Linie. Paint.Cap.ROUND: abgerundeter Anfang und abgerundetes Ende, ein Halbkreis mit dem Zentrum am Anfangspunkt/Endpunkt mit dem Radius der halben Strichbreite. Entspricht der Anmutung, wenn man mit einem runden Filzstift Linien zieht. Paint.Cap.SQUARE: quadratischer Anfang und quadratisches Ende, das Quadrat hat sein Zentrum am Anfangspunkt bzw. Endpunkt und die Höhe der Liniendicke. Der Unterschied zu BUTT ist, dass die Linie damit über den Anfangspunkt und Endpunkt »hinaussteht«.
setStrokeJoin(Paint.Join join)	Hier legen wir fest, wie die Verbindungsstellen von miteinander verbundenen Linien (durch Pfade) gezeichnet werden. Paint.Join.BEVEL: Die Verbindungsstelle ist eine platte, gerade Linie. Paint.Join.ROUND: Die Verbindungsstelle ist abgerundet. Paint.Join.MITER: Die Verbindungsstelle ist eine Gehrung (ein Schrägschnitt der Linienenden).
setStrokeMiter(float miter)	Miter ist die Gehrung zweier aufeinanderstoßender Linien. Bei einem Bilderrahmen oder Türrahmen ist die Gehrung der Schrägschnitt der aufeinanderstoßenden Leisten, in der Regel 45°, da die Leisten in einem 90°-Winkel aufeinanderstoßen. Mit setStrokeMiter beeinflussen wir, welche Gehrung benutzt wird, wenn der Winkel der aufeinanderstoßenden Linien ein spitzer Winkel, also kleiner als 90° ist. Dieser Wert ist das Limit, wie weit die Spitze gegenüber dem Anschlusspunkt überstehen darf, um eine Gehrung zu bilden. Würde die Spitze über das Limit hinausgehen, erhält man einen abgeflachten Anschluss der Linien.
setStrokeWidth(float width)	Setzt die Strichbreite der Linien. Ein Wert von 0 zeichnet eine Haarlinie, die immer 1 Pixel breit ist. Ansonsten handelt es sich um geräteabhängige Pixel (in Verbindung mit der Transformationsmatrix). Ein Wert von 1.0 bei einer Skalierung von 1.0 entspricht wiederum einem Gerätepixel.

Tabelle 4.1: **Wichtigste Methoden des Paint-Objekts**

setAntiAlias(boolean)	Bei true werden beim Zeichnen Aliaseffekte (Treppchenbildung) vermieden. Antialiasing wird nur auf die Kanten der Figuren angewendet, nicht auf den Inhalt.
setShadowLayer(float radius, float dx, float dy, int color)	Fügt eine Schattenebene hinzu. Dadurch erhält jede Figur einen Schatten mit dem angegebenen Radius und dem Versatz dx und dy. Die Basisfarbe wird in color angegeben. Der Radius bestimmt die Schärfe des Schattens. Ein Radius von 0 löscht die Schattenebene.
clearShadowLayer()	Löschen der Schattenebene.
setTypeFace(Typeface typeface)	Bestimmt Schriftart und Schriftstil zum Zeichnen von Text. Die eingebauten Standardfonts sind: Typeface.DEFAULT: Normale Standardschrift. Typeface.DEFAULT_BOLD: Normale Standardschrift für Fettdruck. Typeface.MONOSPACE: Standardschrift mit fester Laufweite Typeface.SANS_SERIF: Standardschrift ohne Serifen (ähnlich Arial) Typeface.SERIF: Standardschrift mit Serifen (ähnlich Times Roman)

Tabelle 4.1: **Wichtigste Methoden des Paint-Objekts (Forts.)**

Neben diesen Methoden liefert das Paint-Objekt weitere Methoden zur Textberechnung und für weitere Effekte. Hier ist Experimentieren angesagt.

Ein Beispiel für die Verwendung des Canvas, des Paint-Objekts und des Path-Objekts ist der ColorChooser im ScrapBook.

Der ColorChooser stellt ein Farbrad dar, das sich am HSV-Farbmodell orientiert, wobei keine »unendliche« Zahl der Farben angeboten wird, sondern, im Beispiel, die Farben im Abstand von 6° auf dem Farbkreis, was 60 Farben ergibt. Im ColorChooser werden verschiedene Techniken benutzt, unter anderem die Technik, das Farbrad nicht ständig neu zu zeichnen, die Konvertierung von HSV-Farben in RGB-Farben sowie die Darstellung der Farbradsegmente als Path.

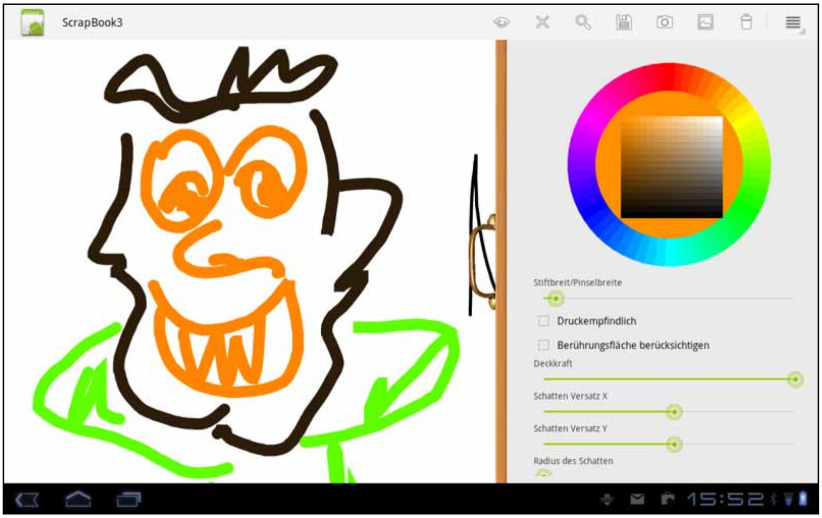


Abbildung 4.1: **ScrapBook mit ColorChooser**

Wir leiten solche Widgets, wie in Kapitel 3 besprochen, im Grunde direkt von der View ab und implementieren das Zeichnen innerhalb von `onDraw(…)` selbst.

```
public class ColorChooser extends View {
[…]
@Override
public void onDraw(Canvas canvas)
{
    super.onDraw(canvas);
    canvas.save();
    canvas.translate(midX,midY);
```

Mit `canvas.save()` sichern wir den aktuellen Zustand des Canvas. Danach können wir beliebige Transformationen anwenden und später den alten Zustand wieder zurückladen. Hier führen wir eine Translation des Koordinatensystems in die Mitte des Widgets aus, das heißt, ab jetzt ist der Ursprung (0,0) in der Mitte.

```
    Paint paint = new Paint();
    paint.setColor(getCurrentColor());
    canvas.drawCircle(0,0, r0+(segmentWidth/2),paint);
```

Hier zeichnen wir nämlich einen ausgefüllten Kreis mit der gerade gewählten Farbe, bevor wir nachher das Farbrad darum herum zeichnen. Da wir das Koordinatensystem in die Mitte verschoben haben, können wir den Kreis einfach um den Punkt (0,0) herum zeichnen. Das Paint-Objekt wurde vorher mit der aktuellen Farbe besetzt.

```
    canvas.restore();
```

Hier folgt das Restore, um die Verschiebung wieder rückgängig zu machen. Jetzt liegt der Ursprung wieder links oben.

```
    createCachedBitmap(false);
    canvas.drawBitmap(cacheBitmap, 0,0, paint);
```

Es wird, falls erforderlich, das Farbrad erzeugt und dann als Bitmap gezeichnet. Warum tun wir das? Das Farbrad beinhaltet wie im Screenshot zu sehen auch die jeweiligen Farbabstufungen in Sättigung und Helligkeit als Matrix. Diese Matrix muss immer aktualisiert werden, wenn eine andere Farbe auf dem Rad gewählt wurde. Wenn wir sowohl das Rad als auch diese Matrix innerhalb von onDraw immer wieder neu aufbauen, dann ist das nicht so performant. Hier setzen wir dann die Doppelpufferungstechnik ein und erzeugen nach einer Änderung erst einmal eine neue Bitmap, die dann hier auf den Schirm gebracht wird.

Listing 4.2: **onDraw(...)-Methode des ColorChooser**

```
}
[…]
}
```

Das Farbrad und die Matrix selbst werden in einer eigenen Methode erzeugt. Es ist eine empfehlenswerte Technik, das Zeichnen immer in eine eigene Methode auszulagern, denn dann können wir später diese Methode immer wieder verwenden, wo wir auf einen Canvas zeichnen wollen, z.B. auf eine Bitmap, in ein Picture, auf eine Surface-View etc.

```
protected void createCachedBitmap(boolean force)
{
if (cacheBitmap==null || force)
{
    if (cacheBitmap!=null)
    {
        cacheBitmap.recycle();
    }
    cacheBitmap = Bitmap.createBitmap(getWidth(),getHeight(),Bitmap.Config.
ARGB_8888);
    Canvas canvas = new Canvas(cacheBitmap);
    drawChooser(canvas);
}
}
```

Listing 4.3: **Erzeugen der Bitmap mit dem Farbrad**

Die eigentliche Logik befindet sich dann in drawChooser(...), in dem nun das Farbrad und die Helligkeits-/Sättigungsmatrix erzeugt wird.

```
protected void drawChooser(Canvas canvas)
{
    Paint paint = new Paint();
    paint.setAntiAlias(true);
    paint.setStyle(Paint.Style.FILL_AND_STROKE);
    canvas.translate(midX,midY);
```

```
double currentAngle = 0.0;
float[] hsv = new float[3];
float[] hsv0 = new float[3];

Color.RGBToHSV(Color.red(currentColor), Color.green(currentColor), Co➥
lor.blue(currentColor), hsv0);
```

Hier folgt die Schleife, die einen Vollkreis im Farbrad beschreibt, mit der Schrittweite von 6° bzw. der Schrittweite, die wir im ColorChooser definiert haben. Erst einmal erstellen wir ein Segment des Farbrades als Path, diese Methode beschreiben wir etwas später.

```
Path path = createPath(0,r0);
canvas.save();
float angledegrees = (float)Math.toDegrees(this.angle);
```

Durch die Art und Weise, wie ich die Methode zum Erstellen des Kuchenstücks implementiert habe, liegt das Kuchenstück für die 0° (Rot) des Farbkreises im Koordinatensystem rein rechnerisch um 90° rotiert. Deshalb rotiere ich den Canvas hier um 90° nach links, damit das rote Segment »oben« startet.

```
canvas.rotate((float)-90.0f);
while (currentAngle<2*Math.PI)
{
    float degrees = (float)Math.toDegrees(currentAngle);
    hsv[0]=degrees;
    hsv[1]=1.0f;
    hsv[2]=1.0f;

    int color = Color.HSVToColor(hsv);
    paint.setColor(color);
```

Hier wird nun das Kuchenstück in der entsprechende Farbe gezeichnet …

```
    canvas.drawPath(path, paint);
```

… und der Canvas um den Winkel unseres Kuchenstücksegments weiterrotiert.

ACHTUNG

Die Matrix wird immer mit dem hier angegebenen Wert kombiniert! Der Canvas wird tatsächlich weiterrotiert und nicht etwa auf den angegebenen Wert gekippt.

```
    canvas.rotate((float)angledegrees);
    currentAngle+=this.angle;
}
```

Der Canvas wird wieder auf die Ursprungswerte gesetzt, damit werden auch alle Matrizen wieder zurückgesetzt, und zwar auf den Zustand, den sie beim Aufruf von `save()` hatten.

```
    canvas.restore();
    int step = 10;
    int w = (int)(r0*0.66f);
    float scalex = step*1.0f/(2*w);
    float scaley = step*1.0f/(2*w);
    Rect r = new Rect();
    float s = 1.0f;
```

Hier wird nun die Sättigungs- und Helligkeitsmatrix der aktuellen Farbe in den Farbkreis gemalt. Auf der X-Achse wird die Sättigung abgetragen, auf der Y-Achse die Helligkeit.

Listing 4.4: **Erstellen des Farbrads und der Farbmatrix**

```
    for (int x=(int)(midX - w); x<midX+w; x+=step)
    {
        float v = 1.0f;
        for (int y=(int)(midY - w); y<midY+w; y+=step)
        {
        hsv[0]=hsv0[0];
        hsv[1]=s;
        hsv[2]=v;
        int color = Color.HSVToColor(hsv);
        paint.setColor(color);
        r.set(x-midX, y-midY, x-midX+step, y-midY+step);
        canvas.drawRect(r, paint);
        v-=scaley;
    }
    s-=scalex;
    }
}
```

Das Kuchenstück selbst wird wie folgt erstellt. Es werden die Eckpunkte des Segments im Abstand `r0` vom Koordinatenursprung um den Winkel `startAngle` herum berechnet und ein geschlossenes Polygon als Pfad erstellt. »Historisch« konnte ich diese Funktion benutzen, um für alle Segmente einzelne Pfade zu erstellen; wie oben gesehen benutzen wir den Pfad nun aber einfach mehrfach, indem das Koordinatensystem gedreht wird.

Listing 4.5: **Erstellen eines Segments des Farbrads**

```
protected Path createPath(double startAngle, double r0)
{
    float[] x = new float[4];
    float[] y = new float[4];

    double halfAngle = this.halfAngle+Math.toRadians(0.5);

    x[0] = (float)(r0*Math.cos( startAngle-halfAngle ));
    y[0] = (float)(r0*Math.sin( startAngle-halfAngle ));

    x[1] = (float)(r0*Math.cos( startAngle+halfAngle ));
    y[1] = (float)(r0*Math.sin( startAngle+halfAngle ));

    x[2] = (float)((r0+segmentWidth)*Math.cos( startAngle+halfAngle ));
    y[2] = (float)((r0+segmentWidth)*Math.sin( startAngle+halfAngle ));
```

```
    x[3] =  (float)((r0+segmentWidth)*Math.cos( startAngle-halfAngle ));
    y[3] = (float)((r0+segmentWidth)*Math.sin( startAngle-halfAngle ));

    Path segmentPath = new Path();
    segmentPath.moveTo(x[0], y[0]);
    segmentPath.lineTo(x[1], y[1]);
    segmentPath.lineTo(x[2], y[2]);
    segmentPath.lineTo(x[3], y[3]);
    segmentPath.lineTo(x[0], y[0]);
    segmentPath.close();
    return segmentPath;
}
```

Der ColorChooser ist eine Komponente innerhalb des ScrapBooks, um die Farbe für das aktuelle Scribbling auszuwählen. Im Screenshot sieht man in Form der aufgezogenen Schublade noch weitere Widgets, um Stiftbreite und andere Parameter wie den Schatten zu setzen.

INFO

Die aktuelle Version kann noch nicht, bzw. nicht mehr, mit den Optionen »Druckempfindlich« und »Berührungsfläche berücksichtigen« umgehen. Die Werte werden zwar aufgezeichnet, aber noch nicht im Scribble umgesetzt. Hier ist noch etwas Feinarbeit vonnöten, um die Zeichenoperationen entsprechend auszufeilen. Der vorherige Ansatz hat nämlich einfach Kreise für jeden Scribble-Punkt gezeichnet und hier den Radius und den Alpha-Wert vom Druck und von der Berührungsfläche abhängig gemacht. Damit sind allerdings keine schönen Linienzüge zu realisieren, deshalb ist dieses Feature zurzeit außer Kraft.

Im ScrapBook ist das Zeichnen der Scribbles mit ähnlichen Techniken realisiert. Durch den TouchEvent-Listener werden die Punkte an das Scribble übergeben und dort in Pfade umgesetzt. Die aktuelle Farbe (aus dem ColorChooser) sowie die Deckkraft und Einstellungen für den Schatten (aus der Schublade) werden innerhalb der draw(...)-Methode der Klasse `Scribble` berücksichtigt.

```
public void draw(Canvas canvas)
{
    canvas.save();
    Paint paint = new Paint();
    paint.setAntiAlias(true);
    paint.setDither(true);
```

Im Folgenden werden die jeweiligen Parameter verwendet, die von außen im aktuellen Scribble gesetzt oder durch das Laden des Scribbles wieder hergestellt wurden.

```
    paint.setColor(currentParameters.getColor());
    paint.setAlpha(currentParameters.getAlpha());
    paint.setShadowLayer(currentParameters.getShadowlayer_radius(), current➥
    Parameters.getShadowlayer_x_offset(), currentParameters.
getShadowlayer_y_offset(), currentParameters.getShadowcolor());
    paint.setStrokeWidth(currentParameters.getStrokeWidth());
```

```
    paint.setStyle(Paint.Style.STROKE);
    paint.setStrokeJoin(Paint.Join.ROUND);
    paint.setStrokeCap(Paint.Cap.ROUND);
```

Und nun wird der Pfad erzeugt.

```
    Path path = new Path();
    for (int i : penStrokes.getPenIds())
    {
    […]
        if (ppen!=null && (ppen.x!=pen.x || ppen.y!=pen.y))
        {
        path.lineTo(pen.x, pen.y);
        }
        else
        {
        path.moveTo(pen.x, pen.y);
        }
        ppen = pen;
    }
```

Und hier wird der Pfad auf den Canvas gebracht.

Listing 4.6: **Zeichnen eines Scribbles auf einem Canvas**

```
    canvas.drawPath(path,paint);
}
}
canvas.restore();
}
```

Auch hier ist das Zeichnen aus der eigentlichen onDraw(…)-Methode des ScribbleWidgets ausgelagert, um beim Malen der Scribbles wiederum Optimierungen durchführen zu können. Auch das ScribbleWidget benutzt z.B. die Doppelpufferungstechnik, um das gesamte Scribble (das aus vielen einzelnen Scribble-Objekten bestehen kann) performant anzuzeigen.

Neben diesem Aspekt ist es ja auch so, dass das Scribble genau weiß, wie es sich zeichnen muss. Vom objektorientierten Ansatz her müssen wir also die draw(…)-Methode hier ansiedeln. Das Scribble-Objekt dient hier nämlich als Basisobjekt für weitere Scribble-Objekte, z.B. um Text in das ScrapBook aufnehmen zu können oder auch andere geometrische Formen bereitzustellen.

Der folgende Auszug zeigt ein solches Textobjekt. Der Clou des Textobjekts ist, dass man den Text praktisch mit dem Finger oder einem Stift hinzeichnen kann, der eingegebene Text folgt dem gezeichneten Pfad. Allerdings benutzen wir hier nicht alle Pfadpunkte, sondern nur den ersten, den mittleren und den letzten Pfadpunkt und legen durch diese drei Punkte eine Kurve, entlang der der Text dann gezeichnet wird. Entscheidend ist die Methode `Canvas.drawTextOnPath(…)`, die wiederum zum Experimentieren einlädt.

```
public class TextScribble extends Scribble {
private CharSequence text = "";
[…]
@Override
public void draw(Canvas canvas)
{
    canvas.save();
    […]

    paint.setColor(getCurrentParameters().getColor());
    paint.setAlpha(getCurrentParameters().getAlpha());
    paint.setShadowLayer(getCurrentParameters().getShadowlayer_radius(),
getCurrentParameters().getShadowlayer_x_offset(), getCurrentParameters().
getShadowlayer_y_offset(), getCurrentParameters().getShadowcolor());
```

Die Textgröße wird im Verhältnis zur gewählten Strichbreite gesetzt. Alternativ können wir auch im Dialog noch die Textgröße festlegen lassen.

```
    paint.setTextSize(5.0f*getCurrentParameters().getStrokeWidth());
    […]
    if (strokes.getCount()==1)
    {
```

Wenn nur ein Stützpunkt vorhanden ist, etwa durch ein einfaches Tippen auf den Touchscreen, zeichnen wir den Text ab diesem Startpunkt.

```
        Pen pen = strokes.getStrokes().get(0);
        canvas.drawText(text.toString(), pen.x, pen.y, paint);
    }
    […]
    List<Pen> pens = strokes.getStrokes();
    int end = pens.size()-1;
    int mid = end/2;
    if (strokes.getCount()>=2)
    {
```

Wenn mehrere Punkte vorhanden sind, dann ziehen wir eine Linie (wenn es nur zwei sind) oder aber eine Kurve.

```
        path.moveTo(pens.get(0).x, pens.get(0).y);
        if (strokes.getCount()>=3)
        {
        path.quadTo(pens.get(mid).x, pens.get(mid).y,pens.get(end).x, pens.
get(end).y);
        }
        else
        {
        path.lineTo(pens.get(end).x, pens.get(end).y);
        }
    }
    […]
```

Und hier wird der Text gezeichnet. Falls die Touch-Operationen noch andauern, dann wird der Pfad als Hilfestellung auch gezeichnet; wenn die Zeichenoperation beendet ist, dann wird nur der Text gezeichnet.

Listing 4.7: **Ein Textobjekt**

```
    if (!strokes.isFinished()) canvas.drawPath(path, paint0);
    canvas.drawTextOnPath(text.toString(), path, 0, 0, paint);
}
break;
}
}
[…]
}
```

Im Scribble-Objekt des ScrapBooks spielt das Path-Objekt eine große Rolle. Das Path-Objekt ist ideal, um aus grundlegenden Zeichenoperationen komplexere Pfade (daher der Name) zu erzeugen. Das können einfache Linienzüge (Polygone) sein, aber in Verbindung mit Bézierkurven, Kreisbögen, Ellipsen und Rechtecken können auch komplexe Figuren erzeugt werden, die dann entweder als Kontur, gefüllte Figur oder wie in diesem Fall auch als Pfad für Textoperationen dienen können.



Abbildung 4.2: **Das Textobjekt in Aktion**

Die Klassen `Scribble`, `ScribbleWidget` und weitere Klassen aus dem ScrapBook-Projekt beinhalten noch weitere Anregungen für das Zeichnen von 2D-Grafik.

4.1.3 SurfaceView

Im vorigen Abschnitt haben wir uns mit dem Zeichnen auf einem Canvas in Verbindung mit einem eigenen Widget (dem ScribbleWidget) auseinandergesetzt. Die Aktualisierung des Bildschirms mit dem Inhalt des Canvas, der per `onDraw(…)` gezeichnet wird, wird durch

das View-System kontrolliert und immer im UI-Thread unserer Anwendung ausgeführt. Das passiert immer automatisch, wenn sich die View-Hierarchie ändert oder Teile unserer View verdeckt oder aufgedeckt werden. Oder wir weisen eine Aktualisierung mittels `invalidate(…)`, besser über `postInvalidate(…)` an. Wir haben aber keine Garantie dafür, wann die Aktualisierung ausgeführt wird. Die Anforderung wird in die Warteschlange im UI-Thread eingereiht und dann abgearbeitet, wenn unsere Anwendung die entsprechende Zeit zum Abarbeiten der Nachrichten hat.

Es ist offensichtlich, dass diese Vorgehensweise für grafische Darstellungen, die sich nicht häufig oder schnell ändern und wenige Animationen beinhalten, gut geeignet ist, also z.B. für die normalen Widgets oder auch für die Leinwand des ScrapBooks.

Es gibt aber Anwendungen, die den Bildschirminhalt schneller und direkt aktualisieren müssen. Das ist z.B. die Kameravorschau, aber auch das Abspielen von Videos oder Spiele, die extrem animationslastig sind.

Um solche Anwendungen mit dem Framework zu realisieren, bietet Android die SurfaceView an. Die SurfaceView erlaubt quasi das direkte Zeichnen auf den Bildschirm, und die Aktualisierung der SurfaceView kann innerhalb von Threads erfolgen, ohne dass auf die Rückkehr zum UI-Thread gewartet werden muss.

Damit können wir z.B. schon kleine Spiele bauen oder im Falle des ScrapBooks die Kameraansicht direkt in unser Programm einbinden.

Wichtig ist, dass wir unsere Widgets, die wir auf eine SurfaceView zeichnen wollen, von der SurfaceView ableiten. Diese Klasse beinhaltet die Funktionalität, einen SurfaceHolder anzufordern, über den dann die Zeichenoperationen abgewickelt werden können.

```
public class CameraView extends SurfaceView implements SurfaceHolder.Call➥
    back,
Camera.PreviewCallback,
Camera.ErrorCallback,
Camera.AutoFocusCallback,
Camera.OnZoomChangeListener,
Camera.PictureCallback,Camera.ShutterCallback {
[…]
private void initView()
{
    camera = null;
```

Hier sehen wir, wie der SurfaceHolder angefordert und parametrisiert wird. `setType(…)` sollte laut Doku nicht mehr notwendig sein, auf manchen Systemen stürzt die Preview allerdings ab, wenn hier nicht der vorliegende Typ gesetzt wird.

Die Kommunikation mit der Surface findet dann über den SurfaceHolder statt, dieser stellt Methoden bereit, um z.B. einen Canvas zum Zeichnen anzufordern und den Canvas zur Surface zu schicken. Der Holder übernimmt auch die Erstellung der Surface, wenn unsere View die Surface benötigt.

```
    surfaceHolder = getHolder();
    surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS); //Laut
Doku "deprecated", muss aber in diesem Kontext gesetzt werden, sonst gibts
Probleme mit dem Preview (Absturz)!!
    surfaceHolder.addCallback(this);
}
[…]
```

Um mitzukriegen, wann die Surface zur Verfügung steht oder wann sie nicht mehr zur Verfügung steht, implementieren wir die folgenden Callbacks.

```
public void surfaceChanged(SurfaceHolder holder, int format, int width, int
height) {
    Log.d(Globals.LOG_TAG, "CameraView.surfaceChanged()");
}
```

Wenn die Surface erstellt wurde und zur Verwendung bereitsteht, dann wird `surfaceCreated(…)` aufgerufen. Ab jetzt können wir auf die Surface zeichnen, hier übergeben wir den Holder einfach an die Kamerapreview. Die Kamera, in einem separaten Thread laufend, wird dann die Vorschau auf unsere Surface bringen.

```
public void surfaceCreated(SurfaceHolder holder) {
    try {
    openCamera();
    if (camera!=null)
    {
        Log.d(Globals.LOG_TAG, "CameraView.surfaceCreated()");
        camera.setPreviewDisplay(holder);
        camera.setPreviewCallback(this);
        setState(STATE_INITIALIZED);
    }
    else
    {
    setState(STATE_INITIALIZATIONFAILED);
    }
    } catch (IOException e) {
    Log.d(Globals.LOG_TAG, "camera.setPreviewDisplay(holder)",e);
    }
}
```

Wenn die Surface zerstört wird, werden wir über surfaceDestroyed darüber informiert. Ab jetzt sind keine Operationen mehr auf der Surface möglich, hier informieren wir die Kamera davon und stoppen die Kameravorschau. Die Surface wird z.B. zerstört, wenn die Anwendung schlafen gelegt wird, weil sich z.B. eine andere Anwendung komplett darüberlegt.

Listing 4.8: **SurfaceView für Kamerabildvorschau**

```
public void surfaceDestroyed(SurfaceHolder holder) {
    try {
    Log.d(Globals.LOG_TAG, "CameraView.surfaceDestroyed()");
    if (camera!=null)
    {
```

```
            camera.setPreviewCallback(null);
            camera.setPreviewDisplay(null);
            camera.release();
            camera = null;
            surfaceHolder = null;
        }
        } catch (IOException e) {
            Log.d(Globals.LOG_TAG, "camera.setPreviewDisplay(holder)",e);
        }
}
[…]
}
```

Dieses Widget benutzen wir im ScrapBook, um direkt auf das Kamerabild zu zeichnen. Das ist hier im experimentellen Stadium und erst einmal einfach nur ein Gag, demonstriert aber sehr schön die Möglichkeiten der SurfaceView, aber auch die Möglichkeiten, durch geschicktes Überlagern von Widgets interessante Effekte zu erzielen:

```
public void toggleCameraBackgroundView(boolean on)
{
if (on)
{
    getScribbleWidget().setBackgroundImageContentUri(null);
    getScribbleWidget().setScaleFactor(1.0f);
    getScribbleWidget().setTranslucent(true);
    if (cameraView == null)
    {
```

Hier passiert das Entscheidende. Die CameraView wird über den LayoutInflater geladen und in unser Hauptlayout eingehängt. Die View R.id.scribble_widget_live_background ist ein einfaches FrameLayout, das nun die CameraView aufnimmt und durch die Deklaration in unserer Layoutressource **unter** der ScrapBook-Leinwand liegt. Mittels startPreview() wird die Kameravorschau aktiviert, und die Kamera bringt das Vorschaubild auf die entsprechende SurfaceView, die durch unser Scribble hindurchscheint.

Listing 4.9: **Aktivieren der Kameravorschau als überlagerte View im ScrapBook**

```
        ViewGroup root = (ViewGroup)findViewById(R.id.scribble_widget_live_
background);
        getLayoutInflater().inflate(R.layout.camera_view, root);
        cameraView = (CameraView)findViewById(R.id.camera_view);
        }
        cameraView.setVisibility(View.VISIBLE);
        cameraView.startPreview();
    }
    else
    {
    if (cameraView != null)
    {
        cameraView.stopPreview();
        cameraView.releaseCamera();
        cameraView.setVisibility(View.INVISIBLE);
```

```
            getScribbleWidget().setTranslucent(false);
        }
    }
}
```

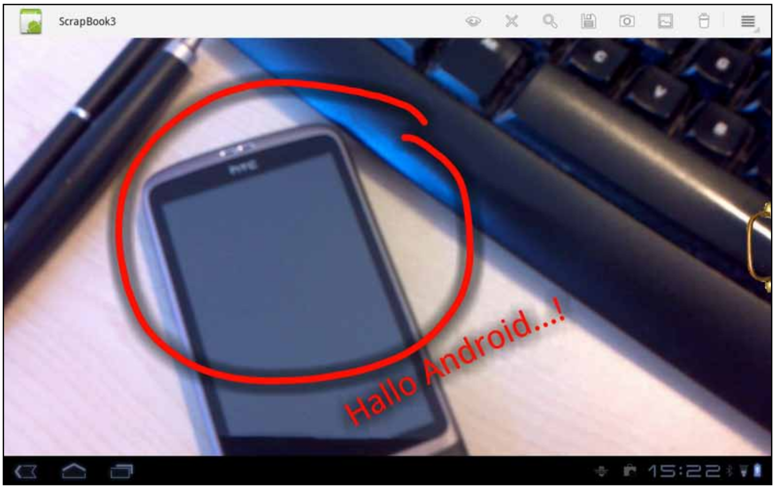


Abbildung 4.3: **Kameravorschau mit überlagertem Scribble**

In der Spielwiese finden wir ein weiteres Beispiel für die Überlagerung des Kamerabildes durch eine Kompassanzeige. Der Schritt nun zu einer eigenen Augmented-Reality-Anwendung ist damit gar nicht mehr weit. Wir werden später auch noch sehen, wie wir Positionsdaten mittels GPS abfragen können.

Nun, das macht doch Lust auf mehr, oder?

Eine weitere Anwendung für die SurfaceView sind Animationen und Spiele. In den Beispielen findet sich das MarbleGame, das ursprünglich dafür gedacht war, die Möglichkeiten der Sensoren für die Steuerung auszuloten und ein wenig Physiksimulation zu betreiben. Daraus hat sich dann ein kleines Experiment entwickelt, bei dem die Bewegung der Murmel, die nun eher ein Ball ist, vor einem animierten Hintergrund stattfindet. Der Ball kann hier durch eine Landschaft springen, die sich in Sandwichtechnik semidreidimensional bewegt. Der Effekt ist dem nachempfunden, was man sieht, wenn man aus dem Fenster eines fahrenden Zugs schaut: Die Landschaft im Hintergrund bewegt sich langsam an uns vorbei, der Mittelteil etwas schneller, und die unmittelbare Szenerie vor uns fliegt nur so vorbei.

Diese Animation des Hintergrunds und des Balls ist ebenfalls mit einer Surface realisiert, da hier die Aktualisierung sehr schnell und direkt vonstattengehen muss.

```
public class GameThread extends Thread implements SurfaceHolder.Callback {
[…]
@Override
public void run() {
    isRunning = true;
    Log.d(Globals.LOG_TAG,this.getClass().getName()+".run()");
```

Unser GameThread kennt den SurfaceHolder und kann also darauf zeichnen. Wir sehen hier eine Schleife, die so lange ausgeführt wird, wie das Flag isRunning auf true gesetzt ist. Das Flag wird von außen gesteuert. Auch der Status des Threads wird von außen gesteuert, damit er schlafen gelegt werden kann, wenn die Anwendung ebenfalls pausiert.

ACHTUNG

In diesem Beispiel lasse ich den Thread aber weiterlaufen, es wird nur nichts weiter gemacht, als ständig die Schleife zu durchlaufen. Das ist in der Realität nicht optimal, da der Thread trotzdem Ressourcen verbraucht. Besser ist es, den Thread zu zerstören und beim Aufwecken des Spiels wieder zu initialisieren.

```
    while (isRunning) {
        if (surfaceState == SURFACE_AVAILABLE) {
        Canvas c = null;
        try {
```

Mittels lockCanvas(...) fordern wir eine Leinwand zum Zeichnen an, die mit der Surface verknüpft ist.

```
        c = surfaceHolder.lockCanvas(null);
        synchronized (surfaceHolder) {
```

Indem wir mit dem Holder synchronisieren, ist es potenziell möglich, auch von mehreren Threads auf die Surface zuzugreifen. Die folgenden Zeilen sind das Entscheidende. updatePhysics() aktualisiert die Ballposition abhängig von Geschwindigkeit und Neigung des Geräts, und doDraw(...) bringt das Ganze auf den Canvas.

```
        if (state == STATE_RUNNING) updatePhysics();
        doDraw(c);
        }
    } finally {
    // do this in a finally so that if an exception is thrown
    // during the above, we don't leave the Surface in an
    // inconsistent state
    if (c != null) {
```

Mit dieser Methode wird der Bildschirm nun letztendlich aktualisiert.

```
    surfaceHolder.unlockCanvasAndPost(c);
    }
    }
    }
    }
    Log.d(Globals.LOG_TAG,this.getClass().getName()+".run() - finished");
}
```

Die Methode doDraw(...) benutze ich hier, um das aktuelle Spielfeld zu zeichnen. Die Layer des Spielfeldsandwichs werden in einer eigenen Klasse abgehandelt, und der Ball zeichnet sich auch selbst.

Listing 4.10: **Anwendung einer Surface in einem Thread**

```
private void doDraw(Canvas c) {

if (state > STATE_UNINITIALIZED)
{
c.drawRect(r, paint);

if (backgroundLayer!=null) backgroundLayer.draw(c);
if( background2Layer!=null) background2Layer.draw(c);
if (theBody!=null) theBody.draw(c);
if (foregroundLayer!=null) foregroundLayer.draw(c);
}

}
[…]
class Body {
[…]
public void draw(Canvas canvas)
{
if (drawable==null)
{
    canvas.drawCircle(position.getX() - environment.getWindow().left, posi➥
    tion.getY()- environment.getWindow().top, getRadiusY(), paint);
}
else
{
    canvas.save();
    canvas.rotate(rotation,position.getX() - environment.getWindow().left,
(int)position.getY()- environment.getWindow().top);
    drawable.setBounds((int)position.getX() - environment.getWindow().left -
getRadiusY(), (int)position.getY()- environment.getWindow().top - getRadi➥
    usY(), (int)position.getX() - environment.getWindow().left + getRadi➥
    usY(), (int)position.getY()- environment.getWindow().top + getRadi➥
    usY());
    drawable.draw(canvas);
    canvas.restore();
}
}
[…]
}
```



Abbildung 4.4: **MarbleGame in Aktion**

Die Hintergründe sind aus Packpapier, Tapete, Pappe und Deko-Materialien hergestellt und einzeln fotografiert. Als transparente PNG-Dateien werden sie in der Anwendung entsprechend übereinandergelegt und aneinandergehängt, um die Anmutung einer endlosen vorüberziehenden Landschaft zu erzielen.

In den Sourcen zum MarbleGame finden sich auch einige Klassen zur Vektorrechnung und zur Anwendung der Bewegungsphysik auf den Ball.

4.1.4 Drawables

Drawables sind Klassen, die irgendetwas zeichnen. Im obigen Beispiel haben wir oft direkt auf den Canvas mit den entsprechenden Primitiven gezeichnet. Allerdings haben wir auch schon häufig die Methoden zum Zeichnen in eigene Klassen (Scribble, TextScribble) ausgelagert und somit etwas Ähnliches wie Drawables erstellt.

Drawables haben nun den Vorteil einer eigenen Abstraktion, und als Drawable liegen einige Klassen vor. Ein großer Vorteil ist, dass alle Drawables in den Ressourcen deklariert werden können, egal ob es sich um Bilder oder um Figuren (ShapeDrawables) oder statusabhängige Drawables oder vieles mehr handelt.

Weiterhin können Drawables komplexere Strukturen abbilden wie AnimationDrawable, TransitionDrawable, LevelListDrawable. Dabei verändert sich das Drawable entweder über die Zeit hinweg oder abhängig von einem Status, einem Level oder anderen Kriterien.

Und: Drawables können z.B. für den Hintergrund von Views eingesetzt werden, und damit ist dann animierten Hintergründen, der Darstellung des Widget-Status mit eigenen Drawables Tür und Tor geöffnet.

Die häufigste Form der Drawables sind die Icons, die wir in der Anwendung verwenden, sowie Hintergründe und 9-Patches. Diese Drawables sind in der Regel PNG-Dateien, die wir mit einem entsprechenden Grafikprogramm entwerfen können.

Nehmen wir als Beispiel noch einmal das MarbleGame. Wir können den etwas langweiligen roten Ball, der einfach als roter Kreis gezeichnet wird, durch ein BitmapDrawable austauschen und etwas sympathischer gestalten.



Abbildung 4.5: **Der Ball für das MarbleGame**

Die Datei speichern wir in `res\drawable-mdpi`. Dadurch legen wir die Basisgröße des Balls, der 64 Pixel im Quadrat misst, auf Bildschirme mit mittlerer DPI-Auflösung fest. Im Profil mdpi entsprechen 160 Pixel einem Zoll, unser Ball ist also ca. 1 cm groß.

Wenn wir nun den Ball auf einem anderen Gerät laden, dann wird das Drawable entsprechend skaliert, sodass der Ball auf jedem Bildschirm ca. 1 cm groß ist.

Die Drawable wird unserem Objekt, das den Ball repräsentiert, zugewiesen. Dazu greifen wir über den Anwendungskontext auf das Ressourcensystem zu. Den Kontext haben wir dem GameThread beim Erstellen des Thread-Objekts mitgeteilt:

Listing 4.11: **Zuweisen des Drawables**

```
public GameThread(Environment environment, SurfaceHolder surfaceHolder, Con➥
    text context, Handler handler)
{
[…]
Drawable drawable  = context.getResources().getDrawable(R.drawable.goldene➥
    kugel_zwinkernd);
theBody.setDrawable( drawable );
if (drawable instanceof AnimationDrawable)
{
    ((AnimationDrawable)drawable).setCallback(this);
    ((AnimationDrawable)drawable).start();
}
[…]
}
```

Wir bereiten hier unseren Ball auch schon für weitere Animationen vor, indem wir darauf prüfen, ob es sich bei dem Drawable um ein AnimationDrawable handelt. Um den Ball noch weiter zu animieren und um von der Anwendung aus mit dem Thread kommunizieren zu können, wird dem GameThread neben dem Kontext ein Handler-Objekt übergeben.

INFO

Das Handler-Objekt stellt eine Schnittstelle zur Nachrichtenwarteschlange der Anwendung bzw. der unterschiedlichen Threads der Anwendung bereit. Über den Handler ist es möglich, eine sichere Kommunikation zwischen den Threads aufzubauen, um z.B. Nachrichten zu Steuerungszwecken auszutauschen. Handler-Objekte und Looper sind wichtige Utensilien für »painless threading«. In unserem Beispiel erhält der GameThread einen Handler, der mit der Nachrichtenwarteschlange des UI-Threads der Activity verknüpft ist. Diesen Handler benötigt der GameThread zur Abwicklung der Animation des Balls, die wir uns gleich noch anschauen.

Anstelle des roten Kreises zeichnen wir nun das Drawable:

Listing 4.12: **Zeichnen des Balls**

```
canvas.save();
canvas.rotate(rotation,position.getX() - environment.getWindow().left, (int)
position.getY()- environment.getWindow().top);
drawable.setBounds((int)position.getX() - environment.getWindow().left -
getRadiusY(), (int)position.getY()- environment.getWindow().top - getRadi➥
    usY(), (int)position.getX() - environment.getWindow().left + getRadi➥
```

```
    usY(), (int)position.getY()- environment.getWindow().top + getRadi➥
    usY());
drawable.draw(canvas);
canvas.restore();
```

Wichtig ist, dass wir die Begrenzung des Drawables korrekt setzen, das ist nämlich der Bereich, in dem der Ball dann auf dem Canvas gezeichnet wird.

Wir können das noch interessanter gestalten, indem wir die Möglichkeit der Rotation mit einbeziehen.

Die Rotation selbst errechne ich in dieser Simulation aus dem Vorzeichen der Geschwindigkeit in X-Richtung und einer festen Rotationsgeschwindigkeit (von ca. 75° Änderung des Winkels pro Sekunde).

Listing 4.13: **Berechnen der Rotation**

```
public void tick()
{
long now = System.currentTimeMillis();
long elapsed = now - t0;
float sElapsed = (float)elapsed/1000f;
[…]
rotation = rotation + Math.signum(v0.getX()) * sElapsed*vrotation;
[…]
}
```

Die eigentliche Rotation findet dann folgendermaßen statt:

Listing 4.14: **Rotation des Canvas**

```
canvas.rotate(rotation,position.getX() - environment.getWindow().left, (int)
position.getY()- environment.getWindow().top);
```

Auch hier sichern wir den Zustand des Canvas vor der Rotation und stellen danach den Zustand wieder her.

INFO

Die Berechnung des Pivotpunkts (um den rotiert wird – hier muss es ja der Mittelpunkt des Balls sein) basiert hier auf der Position des Balls in meiner Spielfeldwelt und dem linken Rand des Fensters auf meine Spielfeldwelt. Ich nutze hier nicht die Translation des Canvas, sondern eine eigene Welt-/Fenster-Konversion. Dasselbe würde aber auch mit einer Translation des Canvas funktionieren.

An diesem Beispiel lässt sich schön erkennen, wozu Drawables dienen können. Wenn wir nun den Ball noch während der Rotation zwinkern lassen wollen, dann können wir dazu ein `AnimationDrawable` verwenden, in dem die verschiedenen Zwinkerstadien verwaltet werden.

Ein AnimationDrawable beinhaltet eine Liste von Drawables, die in einem definierten zeitlichen Abstand angezeigt werden. Diese Art der Animation nennt man *Frameanimation* und basiert auf dem Prinzip des Daumenkinos bzw. auf dem gleichen Prinzip, nach dem traditionell Zeichentrickfilme hergestellt wurden.

Listing 4.15: **Frame-Animation zum Zwinkern der Kugel**

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list
xmlns:android="http://schemas.android.com/apk/res/android"
android:oneshot="false">
    <item android:drawable="@drawable/goldenekugel_mit_augen"
android:duration="2500"/>
    <item android:drawable="@drawable/goldenekugel" android:duration="500"/>
    <item android:drawable="@drawable/goldenekugel_mit_augen"
android:duration="1500"/>
    <item android:drawable="@drawable/goldenekugel" android:duration="250"/>
    <item android:drawable="@drawable/goldenekugel_mit_augen"
android:duration="3000"/>
    <item android:drawable="@drawable/goldenekugel" android:duration="300"/>
</animation-list>
```

Dieses Drawable beinhaltet abwechselnd die Kugel mit offenen Augen und die Kugel mit geschlossenen Augen. Im Attribut `android:duration` wird angegeben, wie lange das jeweilige Frame gültig ist.

Die Standard-Widgets, die von View abgeleitet sind, können direkt mit AnimationDrawables umgehen. Setzen wir den Hintergrund einer View mittels `setBackroundDrawable(…)` auf ein AnimationDrawable, so läuft die Animation nach Aufruf von `AnimationDrawable.start()` automatisch ab.

Damit unser GameThread das auch kann, muss dieser noch das Interface `Drawable.Callback` implementieren:

Listing 4.16: **Implementierung der Callbacks, um AnimationDrawables im Thread zu nutzen**

```
public class GameThread extends Thread implements SurfaceHolder.Callback,
Drawable.Callback {
[…]
@Override
public void invalidateDrawable(Drawable arg0) {
}
@Override
public void scheduleDrawable(Drawable who, Runnable what, long when) {
    if (who == theBody.getDrawable() && what!=null)
    {
        handler.postAtTime( what, who, when );
    }
}
@Override
public void unscheduleDrawable(Drawable who, Runnable what) {
    if (who == theBody.getDrawable() && what!=null)
    {
        handler.removeCallbacks(what,who);
    }
}
[…]
}
```

Durch

Listing 4.17: **Setzen des Callbacks und Starten der Animation**

```
theBody.setDrawable( drawable );
if (drawable instanceof AnimationDrawable)
{
    ((AnimationDrawable)drawable).setCallback(this);
    ((AnimationDrawable)drawable).start();
}
```

wird dann dem Drawable der Callback mitgeteilt und die Animation gestartet. Das Drawable wird dadurch in eine Liste von Animationen aufgenommen, wir müssen allerdings dafür sorgen, dass das Drawable dann an einen Handler gebunden wird, der die Animation innerhalb der Nachrichtenwarteschleife durchrechnet.

INFO

Dadurch benötigen wir für die Animation selbst keinen weiteren Thread, das passiert dann durch das Abarbeiten der Nachrichtenwarteschlange.

Drawables bieten also eine schöne Möglichkeit, etwas, das gezeichnet werden kann, zu kapseln und mannigfaltige Funktionen darin zu implementieren.

Hier haben wir ein PNG-Bild als Drawable benutzt, um zum einen das Laden aus den Ressourcen mit automatischer Skalierung auf unser Ausgabegerät zu nutzen und zum anderen eine Animation zu erstellen.

In Kapitel 3 haben wir auch 9-Patch-Bilder kennengelernt, um dynamische Hintergründe zu erstellen.

Wir können auch selbst Drawables erstellen, indem wir von Drawable ableiten und die `Drawable.draw(Canvas canvas)`-Methode überschreiben, ähnlich der Ableitung eigener Widgets. Eine Anwendung wäre z.B. ein Drawable, das SVG (Scalable Vector Grafics) laden und darstellen kann.

4.1.5 Animationen

Neben der gerade vorgestellten Frame-Animation lassen sich auch komplexere Animationen erstellen. Dazu bietet Android ab Version 3 zwei Möglichkeiten an:

1. View-Animation
2. Property-Animation

View-Animation gibt es bereits seit der ersten Version und zielt speziell darauf ab, Widgets zu animieren, z.B. um ImageViews heraus- und hineingleiten zu lassen. Diese Art der Animation kann man aber nur auf Views anwenden, eigene Objekte mussten bisher durch eigene Strategien animiert werden.

Das Property-Animation-Framework führt nun eine allgemeine Form der Animation ein, die auf beliebige Objekte angewendet werden kann.

TIPP

Die View-Animation ist etwas weniger aufwendig in der Einrichtung und Nutzung. Wenn wir also Views animieren wollen und nur Features der View-Animation nutzen, spricht nichts dagegen, diesen Mechanismus auch weiterhin zu nutzen. Es ist also nicht nötig, alten Code umzuschreiben oder lieb gewonnene Animationen zu ersetzen.

Schauen wir uns kurz die View-Animation an. Anforderung ist hier, nach dem Laden einer Seite in das ScrapBook die aktuelle Seite hinaus- und die neue Seite hineingleiten zu lassen.

Basis dafür ist die `<translate …>`-Animation, mit der wir Verschiebungen in X- und Y-Richtung durchführen können.

Wir benötigen zwei Animationen, eine für das Hinausgleiten und eine weitere für das Hineingleiten:

Listing 4.18: **Animation zum Hinausgleiten**

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@android:anim/accelerate_decelerate_interpolator">
<translate
android:fromXDelta="0.0"
android:toXDelta="-100%"
android:duration="2000"
/>
</set>
```

Listing 4.19: **Animation zum Hineingleiten**

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@android:anim/accelerate_decelerate_interpolator">
<translate
android:fromXDelta="100%"
android:toXDelta="0.0"
android:duration="2000"
/>
</set>
```

Allen Animationen ist in der Regel gemein, dass sie eine Eigenschaft von einem Startwert zu einem Endwert animieren:

NAME	WERT VON	WERT BIS	BEMERKUNG
alpha	android:fromAlpha	android:toAlpha	Transparenz
translate	android:fromDeltaX	android:toDeltaX	Verschiebung der View in X-Richtung um ein Delta

Tabelle 4.2: **Die unterschiedlichen Animationseigenschaften**

NAME	WERT VON	WERT BIS	BEMERKUNG
translate	android:fromDeltaY	android:toDeltaY	Verschiebung der View in Y-Richtung um ein Delta
rotate	android:fromDegrees	android:toDegrees	Rotation um einen Winkel
scale	android:fromXScale	android:toXScale	Skalierung in X-Richtung
scale	android:fromYScale	android:toYScale	Skalierung in Y-Richtung

Tabelle 4.2: **Die unterschiedlichen Animationseigenschaften (Forts.)**

Bei der Rotation und der Skalierung sind noch die Eigenschaften `android:pivotX` und `android:pivotY` wichtig, die den Punkt bestimmen, um den die Rotation bzw. die Skalierung herum stattfindet.

Wie im obigen Beispiel zu sehen, kann man bei den Angaben von Koordinaten oder Koordinatendeltas auch Prozentwerte eingeben. Das ist nützlich, um sich auf die Größe der View zu beziehen, auf die die Animation angewendet wird. Bei der Translation entspricht ein DeltaX-Wert von 100% der Breite der View. Bei der Angabe des Pivotpunkts bedeuten `android:pivotX="50%"` und `android:pivotY="50%"` der Mittelpunkt der View.

Man kann innerhalb eines `<set>` mehrere Transformationen beschreiben, und ein `<set>` kann wiederum ein `<set>` beinhalten. Dadurch können komplexe Animationen durch das Verbinden einzelner Animationen erzeugt werden. Mit `<set android:ordering=["together" | "sequentially"]>` bestimmen wir, ob die Animationen nacheinander (`sequentially`) oder gemeinsam (`together`, das ist auch der Vorgabewert) ausgeführt werden.

Mit dem Attribut `android:duration` bestimmen wir, wie lange die Animation im Ganzen dauert, also in welcher Zeitspanne der Wert vom Startwert zum Endwert interpoliert wird.

Auf welche Weise der Wert interpoliert wird, legen wir mit den Interpolatoren fest:

```
<set
    android:interpolator="@android:anim/accelerate_decelerate_interpolator">
[…]
</set>
```

Der Interpolator bestimmt, wie die Werte abhängig von der Zeit berechnet werden. Der hier benutzte Interpolator beschleunigt am Anfang und bremst zum Ende hin ab.

Ein schöner Effekt lässt sich mit dem OverShootInterpolator erzielen. Dabei schießt die Animation quasi über das Ziel hinaus und kommt dann wie von einem Gummiband gezogen wieder zurück, um schließlich beim Endwert einzurasten.

```
<set
    android:interpolator="@android:anim/overshoot_interpolator">
[…]
</set>
```

Die Animationen laden wir aus der Ressource, die Anwendung innerhalb von Views oder auf einer View ist denkbar einfach:

Listing 4.20: **Laden und Verwenden von Animationen**

```
fadeInAnimation = AnimationUtils.loadAnimation(getContext(), R.anim.scribb➥
    le_fade_in);
fadeOutAnimation = AnimationUtils.loadAnimation(getContext(), R.anim.scribb➥
    le_fade_out);
[…]
view.startAnimation(fadeOutAnimation);
[…]
view.startAnimation(fadeInAnimation);
```

ACHTUNG

Zwei unterschiedliche Animationen hintereinander auszuführen, muss mit ein wenig Logik durchgeführt werden, denn die Animation wird in einem separaten Thread durchgeführt. Die Aufrufe von startAnimation(…) blockieren nicht, bis die Animation fertig ist.

Um die obigen Animationen korrekt hintereinander ablaufen zu lassen, gibt es zwei Möglichkeiten. Wenn die Dauer der ersten Animation bekannt ist, dann können wir das Starten der zweiten Animation verzögern, indem wir die Startzeit setzen:

```
fadeInAnimation.setStartTime(AnimationUtils.currentAnimationTimeMil-
lis()+5000);
view.setAnimation(fadeInAnimation);
```

Mehr Kontrolle können wir gewinnen, indem wir einen Listener bei der ersten Animation registrieren und das, was wir nach Ablauf der Animation tun wollen, ausführen, wenn die Animation wirklich endet:

Listing 4.21: **Sequenzielle Abarbeitung der Animationen**

```
fadeOutAnimation.setAnimationListener(
new Animation.AnimationListener()
{
    @Override
    public void onAnimationEnd(
    Animation arg0) {
        //Tue alles, was nötig ist, bevor die andere Animation startet
        view.startAnimation(fadeInAnimation);
    }

    @Override
    public void onAnimationRepeat(
    Animation arg0) {
    }
```

```
    @Override
    public void onAnimationStart(
    Animation arg0) {
    }

});
view.startAnimation(fadeOutAnimation);
```

Im Falle des ScrapBooks ist die zweite Variante nötig, da wir nach dem Hinausgleiten erst einmal die Bitmap, die unser aktuelles Bild darstellt, neu erzeugen müssen. Das darf aber erst nach Abschluss der Animation passieren. Wenn die Bitmap neu erzeugt ist, dann holen wir diese neue Darstellung mit der zweiten Animation auf den Schirm.

INFO

Wenn wir beide Darstellungen quasi gleichzeitig und nahtlos heraus- und hereingleiten lassen wollen, dann müssen wir bei diesem Ansatz mit zwei Views arbeiten.

Mit dem View-Animation-Framework lassen sich wirklich interessante Effekte erzielen. Mit den mannigfaltigen Kombinationsmöglichkeiten lässt sich unsere Oberfläche dermaßen aufpeppen, dass dem Benutzer glatt schwindelig werden kann.

Das Property-Animation-Framework bietet uns nun noch weitere Möglichkeiten, die über das Animieren von Views hinausgehen.

Wenn wir in unserem ScrapBook animierte Elemente einfügen möchten, dann bietet sich das Property-Animation-Framework an.

Als Beispiel wollen wir alle Elemente des Blatts einfach mal um ihren geometrischen Schwerpunkt kreisen lassen. Unser ScribbleWidget erhält dazu eine Eigenschaft `rotationEffect` vom Typ `float`. Eigenschaften, die mit dem Property-Animation-Framework manipuliert werden sollen, müssen als sogenannte Getter/Setter realisiert werden:

Listing 4.22: **Getter/Setter für die Eigenschaft rotationEffect**

```
private float rotation = 0.0f;
public void setRotationEffect(float rotation)
{
    if (this.rotation == rotation) return;
    this.rotation = rotation;

    for (Scribble scribble: scribbles)
    {
    scribble.setRotation(rotation);
    }

    updateCache(true);
    postInvalidate();
}

public float getRotationEffect()
{
    return this.rotation;
}
```

Die Eigenschaft `rotationEffect` wird hier so implementiert, dass einfach alle Elemente diese Rotation zugewiesen bekommen und danach die Ansicht des Bilds aktualisiert wird.

In der Activity starten wir eine Animation, die diese Eigenschaft animiert:

Listing 4.23: **Starten der Property-Animation**

```
protected void startScribbleAnimation()
{
    ValueAnimator anim = ObjectAnimator.ofFloat(getScribbleWidget(), "rota-
tionEffect", 0f, 360.0f);
    anim.setDuration(5000);
    anim.start();
}
```

Über den `ObjectAnimator` wird ein `ValueAnimator` erzeugt, der auf dem Objekt `getScribbleWidget()` die Eigenschaft `rotationEffect` animiert.

Die Animatoren des Property-Animation-Frameworks bieten ähnliche Eigenschaften wie das View-Animation-Framework. So können Interpolatoren gesetzt, die Dauer bestimmt und per `AnimatorSet`-Objekten komplexe Animationen zusammengebaut werden:

```
anim.setInterpolator(new OvershootInterpolator(2.0f));
```

Property-Animationen können ebenfalls in XML-Ressourcen abgelegt werden. Das Äquivalent zur obigen programmtechnischen Ausführung ist:

Listing 4.24: **Property-Animation aus Ressource**

```
<?xml version="1.0" encoding="utf-8"?>
<set
xmlns:android="http://schemas.android.com/apk/res/android"
android:interpolator="@android:anim/overshoot_interpolator">
<objectAnimator
android:propertyName="rotationEffect"
android:duration="7000"
android:valueTo="360.0"
android:valueType="floatType"/>
</set>
```

Listing 4.25: **Verwenden der Animationsressource**

```
AnimatorSet set = (AnimatorSet) AnimatorInflater.loadAnimator(this,
R.animator.scribble_widget_animator);
set.setTarget(getScribbleWidget());
set.start();
```

Die größte Flexibilität bietet das Property-Animation-Framework, da es aufgrund seiner Konzeption auf alle Objekte angewendet werden kann, die per Getter/Setter Eigenschaften definieren. Der Fantasie sind damit kaum Grenzen gesetzt, Anwendungen mit tollen Effekten auszustatten, sei es auf Ebene der Views und Widgets oder bis zu unserer grafischen Ausgabe auf den Canvas.

4.2 Storage

In Kapitel 3 haben wir uns schon ausführlich mit den Content Providern beschäftigt. Content Provider sind das Mittel der Wahl, um strukturierte Daten in einer SQLite-Datenbank abzulegen.

Das ScrapBook benutzt ebenfalls einen Content Provider, um die einzelnen Seiten in einer SQLite-Datenbank abzulegen. Die Scribbles selbst werden als JSON-Objekte (JavaScript Object Notation) in einem BLOB-Feld (Binary Large Object Block) der ScrapBook-Tabelle abgelegt. Das JSON-Format benutze ich hier, um eine plattformneutrale Speicherung meiner Scribbles zu erreichen, die ich auch einfach über das Netz verschicken und damit auch einen einfachen Synchronisierungsmechanismus bauen kann. Das JSON-Format ist ein kompaktes Datenformat in Textform, das sowohl vom Mensch als auch maschinell sehr einfach gelesen und verarbeitet werden kann. JSON-Objekte sind z.B. immer gültige JavaScript-Objekte, sodass eine Weiterverarbeitung in einer auf JavaScript basierenden Sprache ohne zusätzliche Werkzeuge direkt möglich ist.

Hier sehen wir, wie das aktuelle Scribble gespeichert wird:

Listing 4.26: **Speichern einer ScrapBook-Seite**

```
public void saveTheScribble(final boolean saveAs) throws JSONException
{
    ContentValues values = new ContentValues();
    JSONObject json = new JSONObject();
    saveScribble(json);
    Log.d(Globals.LOG_TAG,json.toString(2));
    values.put(Scrap.Columns.SCRAP_CONTENT,json.toString());

    if (saveAs || contentUri == null)
    {
        Uri uri = getContext().getContentResolver().insert(Scrap.CONTENT_
URI, values);
        contentUri = uri;
        Log.d(Globals.LOG_TAG,uri.toString());
    }
    else
    {

    getContext().getContentResolver().update(contentUri, values, null,
null);
}

createThumbnailFromCachedBitmap(contentUri);
}
```

Die Scribbles selbst wissen, wie sie sich im JSON-Objekt ablegen (und später daraus wieder lesen) sollen:

Listing 4.27: **Speichern der Scribbles im JSON-Objekt**

```
public void saveScribble(JSONObject json) throws JSONException
{
```

```
    if (backgroundImageContentUri!=null)
    {
        json.put("backgroundImageContentUri",backgroundImageContentUri.to➥
    String());
    }
    json.put("count", scribbles.size());
    JSONArray jsonScribbles = new JSONArray();
    for (int i=0; i<scribbles.size(); i++)
    {
        Scribble scribble = scribbles.get(i);
        jsonScribbles.put(scribble.getJSONObject());
    }
    json.put("scribbles", jsonScribbles);
}
```

Das resultierende JSON-Objekt sieht so aus:

Listing 4.28: **Auszug aus der JSON-Darstellung der Scribble-Objekte**


```
{"scribbles":[{"paint.shadowlayer.radius":12,"text":"Hallo
Android...!","paint.shadowlayer.yoffset":2,"paint.alpha":255,"scribbleCreato
r":"TextScribble","paint.shadowcolor":-16119286,"paint.strokewidth":13,"pen-
Strokes":{"multiStrokes":[{"pointerId":0,"penStrokes":{"strokes":[{"pressure
":0,"y":436,"size":0,"x":245.5},{"pressure":0,"y":120,"size":0,"x":590},{"pr
essure":0,"y":125.5,"size":0,"x":927}]}}]},"paint.sizesitive":false,"paint.
pressuresensitive":false,"paint.shadowlayer.xoffset":2,"paint.color":-
33792},…],"count":5}
```


ACHTUNG

Diese Form der Speicherung ist natürlich nicht besonders effizient, da der gesamte JSON-String im Speicher erzeugt wird und eine große Anzahl von Scribbles zum einen den Hauptspeicher beim Laden und Speichern belastet, zum anderen die Performance nicht so hoch ist wie bei einer Speicherung in einem binären, kompakten Format. Für eine spätere produktive Anwendung wäre eine andere Form der Speicherung anzudenken bzw., wenn man bei JSON bleiben möchte, eine speicherschonende Implementierung zu wählen.

Die Ablage in der Datenbank erfolgt per `values.put(Scrap.Columns.SCRAP_CONTENT,json.toString())` und dem anschließenden Aufruf von `getContentResolver.insert(…)`. SQLite bietet eine sehr gute Verwaltung großer Objekte (BLOBs: Binary Large Object Block). Möglicherweise wollen wir aber die Scribbles lieber als eigene Datei ablegen und nur einen Verweis auf diese Datei in unserer Datenbank speichern.

Glücklicherweise bietet uns das Framework einige Methoden, um Dateien zu verwalten.

Dabei ist es wichtig, zwischen drei Speicherorten zu unterscheiden:

1. Interner Speicher
2. Externer applikationsspezifischer Speicher
3. Externer öffentlicher Speicher

INFO

Die Daten einer Anwendung werden gelöscht, wenn diese deinstalliert wird; außer wir speichern die Daten im öffentlichen externen Speicher.

Der interne Speicher ist tatsächlich immer interner Speicher. Die Daten hier sind immer unsere privaten Daten, die nur durch die Anwendung verarbeitet werden können. Andere Anwendungen, oder der Benutzer über USB, haben **keinen** Zugriff auf Daten, die im internen Anwendungsspeicher gespeichert werden. **Ausnahmen** davon sind möglich, indem wir die Dateien mit `MODE_WORLD_WRITEABLE` und/oder `MODE_WORLD_READABLE` öffnen. Dann können andere Applikationen auf die Dateien zugreifen, müssen aber den kompletten Dateinamen kennen. Ich denke, es gibt aber keinen Grund, den Zugriff auf die privaten Dateien zu öffnen. Wenn das notwendig sein sollte, dann sollte das immer in Verbindung mit einer entsprechenden Implementierung eines Content Providers erfolgen.

Der externe Speicher kann applikationsspezifisch oder öffentlich sein. Applikationsspezifisch heißt, dass die Daten bei der Deinstallation auch gelöscht werden, öffentlich heißt hier, dass die Daten Allgemeingut sind und **nicht** gelöscht werden sollen.

Egal ob applikationsspezifisch oder öffentlich, die Daten auf dem externen Speicher können **immer** von allen Anwendungen **gelesen** und durch den User sogar beschrieben werden, wenn das Gerät als USB-Speicher an einen PC angeschlossen wird.

INFO

Externer Speicher bedeutet nicht, dass es sich unbedingt um eine SD-Karte oder Ähnliches handeln muss. Der externe Speicher kann auch fest im Gerät ausgeführt sein. Die Unterscheidung zwischen internem und externem Speicher ist eine logische Aufteilung des Dateisystems.

Um auf den externen Speicher zugreifen zu können, müssen wir im Manifest die Erlaubnis anfordern:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Im ScrapBook benutzen wir den internen Speicher für das Speichern von Thumbnails unserer Scribbles. Diese Thumbnails werden wir später noch in einem App Widget verwenden. Auf den externen Speicher können wir die Scribbles als JPEG-Bilder exportieren und über den MediaScanner auch in die Galerie aufnehmen lassen.

Listing 4.29: **Speichern einer Bitmap als JPEG im internen Speicher**

```
private void createThumbnailFromCachedBitmap(Uri contentUri) {

    int w = cacheBitmap.getWidth();
    int h = cacheBitmap.getWidth();

    int nw = 150;
    int nh = 50;

    float aspect = w/h;
    if (h>w)
```

```
    {
      nh = 150;
      nw = (int)(aspect * nh);
    }
    else
    {
    nw = 150;
    nh = (int)(nw / aspect);
    }

    Bitmap scaledBitmap = Bitmap.createScaledBitmap(cacheBitmap, nw, nh,
true);
    String name = "thubmnail_"+new Long(ContentUris.parseId(contentUri)).
toString()+".jpg";


    try {
        FileOutputStream os = getContext().openFileOutput(name, Context.
MODE_WORLD_READABLE);
        scaledBitmap.compress(Bitmap.CompressFormat.JPEG, 100, os);
        os.close();
    } catch (FileNotFoundException e) {
    e.printStackTrace();
    } catch (IOException e) {
    e.printStackTrace();
    }



    scaledBitmap.recycle();
}
```

Einen Ausgabestream (FileOutputStream) auf den internen Speicher fordern wir mit `Context.openFileOutput(name, Context.MODE_PRIVATE);` an. Mit dieser Methode können aber keine Unterverzeichnisse im internen Speicher erstellt werden. In der Regel sollte das auch nicht nötig sein, da der interne Speicher eigentlich nicht nach außen dargestellt wird und es keinen Grund gibt, Verzeichnisstrukturen zu erstellen.

Folgende Methoden sind im Zusammenhang mit dem internen Speicher nützlich:

FileOutpurStream os = Context.openFileOutput(name, mode) **name** kann keine Pfade beinhalten!	Öffnen oder ersetzen einer Datei im internen Speicher. Folgende Flags können für mode verwendet werden: Context.MODE_PRIVATE : Daten sind privat. Context.MODE_WORLD_READABLE : Daten können von anderen Applikationen gelesen werden. Context.MODE_WORLD_WRITEABLE: Daten können von anderen Applikationen geschrieben werden. Context.MODE_APPEND : Die Datei wird nicht neu erstellt, wenn sie bereits vorhanden ist.

Tabelle 4.3: **Nützliche Methoden im Zusammenhang mit internem Speicher**

File dir = Context.getFilesDir()	Gibt das interne Verzeichnis zurück, in dem die privaten Daten gespeichert werden.
File subdir = Context.getDir(name)	Liefert und erzeugt bei Bedarf ein Unterverzeichnis unterhalb des Datenverzeichnisses der Applikation. Um eine Datei innerhalb des Verzeichnisses zu erstellen, muss dann mit dem File-Objekt gearbeitet werden. **Achtung:** Die Erstellung von Unterverzeichnissen ist nicht typisch und nicht vorgesehen.
Context.deleteFile(name) **name** kann keine Pfade beinhalten!	Löscht eine Datei im Datenverzeichnis der Applikation.
File cacheDir = Context.getCacheDir()	Liefert das Cache-Verzeichnis der Applikation. Die Dateien in diesem Verzeichnis **können** bei Bedarf vom System gelöscht werden, um wieder Speicher freizugeben. **Aber**: Man sollte sich darauf nicht verlassen. Die Anwendung muss selber dafür sorgen, den Cache auf einen gewissen Speicher zu beschränken.

Tabelle 4.3: **Nützliche Methoden im Zusammenhang mit internem Speicher (Forts.)**

TIPP

Um eine Datei im internen Speicher im Cacheverzeichnis anzulegen, gehen wir wie folgt vor:

```
File cacheDir = file = context.getCacheDir();
File cacheFile = new File(cacheDir,"cachefile.dat");
FileOutputStream cacheOs = new FileOutputStream(cacheFile);
cacheOs.write(cacheData);
cacheOs.close();
```

Listing 4.30: Erstellen einer Datei im Cacheverzeichnis

Auf diese Weise können wir auch Unterverzeichnisse per getDir(name) anlegen und Dateien darin verwalten. Allerdings können wir die Berechtigungen dann nicht so ohne Weiteres setzen, und die Dateien sind immer privat.

Den externen Speicher benutzen wir zum Exportieren der Scribbles als JPEG:

Listing 4.31: **Speichern eines Bildes im externen Speicher**

```
public Uri saveJPEG(String bucket, String filename, Bitmap bitmap, boolean
savePublic)
{
    if (!this.isExternalStorageAvailable())
    {
        return null;
    }
    if (!this.isExternalStorageWriteable())
    {
        return null;
```

```
    }

    Uri result = null;

    String path = Environment.DIRECTORY_PICTURES;
    if (bucket!=null && !bucket.equals(""))
    {
        path = path +"/" + bucket;
    }
    File picturePath = null;
    if (savePublic)
    {
        picturePath = Environment.getExternalStoragePublicDirectory(path);
        picturePath.mkdir();
    }
    else
    {
        picturePath = context.getExternalFilesDir(path);
        picturePath.mkdir();
    }
    File file = new File(picturePath, filename );
    OutputStream os;
    try {
        os = new FileOutputStream(file);
        bitmap.compress(Bitmap.CompressFormat.JPEG, 100, os);
        os.close();
        MediaScannerConnection.scanFile(context,new String[] { file.to➥
    String() }, null,null);
        result = Uri.fromFile(file);

    } catch (FileNotFoundException e) {

    } catch (IOException e) {

    }

    return result;

}
```

Hier prüfen wir erst einmal ab, ob der externe Speicher überhaupt verfügbar ist. Wenn das Gerät als Massenspeicher an den PC angeschlossen ist, dann ist das externe Dateisystem nicht mehr für die Anwendung sichtbar, es wurde entladen (unmounted). In diesem Fall können wir nicht auf den externen Speicher zugreifen.

Die Prüfung funktioniert folgendermaßen:

Listing 4.32: **Feststellen des Mount-Status des externen Speichers**

```
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // We can read and write the media
    externalStorageAvailable = externalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
```

```
    // We can only read the media
    externalStorageAvailable = true;
    externalStorageWriteable = false;
} else {
    // Something else is wrong. It may be one of many other states, but all
we need
    //to know is we can neither read nor write
    externalStorageAvailable = externalStorageWriteable = false;
}
```

Mit String `path = Environment.DIRECTORY_PICTURES` holen wir uns den Standardpfad für Bilder und hängen bei Bedarf ein Unterverzeichnis an.

INFO

Die Unterverzeichnisse unter den Medienverzeichnissen bilden in der Galerie die Alben. Im MediaStore sind die Unterverzeichnisse als BUCKET in den Datensätzen gespeichert.

Dann öffnen wir einen Stream und schreiben die Bilddaten in die Datei. Um unsere Daten auch in die Galerie und unser Unterverzeichnis als Album aufnehmen zu lassen, weisen wir den MediaScanner mit `MediaScannerConnection.scanFile(…)`an, unsere Datei in den MediaStore aufzunehmen.

INFO

Das müssen wir hier machen, weil wir ein Unterverzeichnis anlegen. Wenn wir die Bilder direkt im Medienverzeichnis speichern, werden sie automatisch vom MediaScanner aufgenommen.

In der hier vorgestellten Methode können wir entscheiden, ob wir die Dateien öffentlich oder applikationsspezifisch speichern wollen:

Listing 4.33: **Öffentliche oder applikationsspezifische Ablage der Daten**

```
if (savePublic)
{
picturePath = Environment.getExternalStoragePublicDirectory(path);
picturePath.mkdir();
}
else
{
picturePath = context.getExternalFilesDir(path);
picturePath.mkdir();
}
```

Den öffentlichen Pfad erhalten wir per `Environment.getExternalStoragePublicDirectory(path)`, den applikationsspezifischen Pfad per `Context.getExternalFilesDir(path)`.

Wie beschrieben werden die Daten im öffentlichen Pfad bei der Deinstallation der Anwendung nicht gelöscht, was im Falle der exportierten Scribbles sicherlich gewünscht ist.

Folgende Standardverzeichnisse definiert Android im Kontext:

DIRECTORY_ALARMS	Verzeichnis, um Audiodateien für Alarme zu speichern.
DIRECTORY_DCIM	Verzeichnis für Kamerabilder und Videos (DCIM: Digital Camera Images).
DIRECTORY_DOWNLOADS	Verzeichnis, in dem Downloads gespeichert werden sollten.
DIRECTORY_MOVIES	Verzeichnis für Videos (die nicht über die Kamera aufgenommen wurden).
DIRECTORY_MUSIC	Verzeichnis für Musik.
DIRECTORY_NOTIFICATIONS	Verzeichnis für Audiodateien mit Benachrichtigungssignalen.
DIRECTORY_PICTURES	Verzeichnis für Bilder (die nicht über die Kamera aufgenommen wurden).
DIRECTORY_PODCASTS	Verzeichnis für Podcasts.
DIRECTORY_RINGTONES	Verzeichnis für Klingeltöne

Tabelle 4.4: **Standardverzeichnisse**

Diese Verzeichnisse finden sich zum einen unter dem Anwendungshauptverzeichnis und zum anderen unter dem Hauptdatenverzeichnis:

/sdcard/Android/data/<Packagename>/files	Applikationsspezifische Verzeichnisse und Dateien
/sdcard	Öffentliche Verzeichnisse und Dateien

Tabelle 4.5: **Verzeichnisstruktur /sdcard**

Neben den hier genutzten Möglichkeiten, beliebige Dateien zu erstellen, bietet das Framework noch einen interessanten Speicher, die `SharedPreferences`. Diese liegen im internen Speicher, ermöglichen das einfache Abspeichern von Schlüssel-Wert-Paaren und sind damit prädestiniert für das Speichern von Voreinstellungen bzw. aktuellen Einstellungen.

TIPP

Damit bieten sich SharedPreferences auch an, um den aktuellen Zustand der Anwendung über einen langen Zeitraum hinweg zu verwalten. Wir benutzen das hier z.B., um die aktuell ausgewählte Stiftfarbe nicht nur über den normalen Lebenszyklus zu retten, sondern auch dann, wenn wir die Anwendung lange nicht mehr benutzt haben, ein Update installiert oder die Anwendung aus dem Speicher komplett entfernt wurde, um die letzte gewählte Stiftfarbe wieder herzustellen.

Wir fordern ein SharedPreferences-Objekt für unsere Anwendung mittels `Context.getSharedPreferences(name)` oder `Context.getPreferences()` an. Mit dem ersten Aufruf können wir mehrere Dateien verwalten und unterschiedlich benennen, wenn wir nur eine einzige Datei benötigen, benutzen wir den zweiten Aufruf.

Listing 4.34: **Verwenden der SharedPreferences**

```
public void saveState()
{
    SharedPreferences settings = getContext().getSharedPreferences(PREFS_
NAME,Context.MODE_PRIVATE);
    SharedPreferences.Editor editor = settings.edit();
    this.currentParameters.writeToSharedPref(editor);
    editor.commit();
}
public void restoreState()
{
    SharedPreferences settings = getContext().getSharedPreferences(PREFS_
NAME,Context.MODE_PRIVATE);
    this.currentParameters.readFromSharedPref(settings);
    […]
}
[…]
public void writeToSharedPref(SharedPreferences.Editor dest)
{
    dest.putBoolean("prefs_written",true);
    dest.putInt("alpha",alpha);
    dest.putFloat("strokeWidth",strokeWidth);
    dest.putInt("color",color);
    dest.putInt("shadowColor",shadowcolor);
    dest.putFloat("shadowlayer_radius",shadowlayer_radius);
    dest.putFloat("shadowlayer_x_offset",shadowlayer_x_offset);
    dest.putFloat("shadowlayer_y_offset",shadowlayer_y_offset);
}

public void readFromSharedPref(SharedPreferences in)
{
    if (in.contains("prefs_written"))
    {
        alpha = in.getInt("alpha",255);
        strokeWidth = in.getFloat("strokeWidth",12.0f);
        color = in.getInt("color",0);
        shadowcolor = in.getInt("shadowcolor",0);
        shadowlayer_radius = in.getFloat("shadowlayer_radius",0);
        shadowlayer_x_offset = in.getFloat("shadowlayer_x_offset",0);
        shadowlayer_y_offset = in.getFloat("shadowlayer_x_offset",0);
    }
}
```

Wichtig ist, dass das Schreiben der SharedPreferences über das `SharedPreferences.Editor`-Objekt erfolgt. Durch `SharedPreferences.edit()` fordern wir dieses an, und mit `editor.commit()` schreiben wir die Änderungen weg.

Das Schreiben und Lesen verwenden wir z.B. in den Activities oder Fragmenten in `onPause()` und `onResume()`, um den Zustand wegzuspeichern. Der Vorteil gegenüber der Bundles, die wir in `onSaveInstanceState(…)` benutzen können, ist, dass die Einstellungen auch noch lange gültig bleiben.

4.3 App Widgets

App Widgets sind Elemente, die wir auf den Homescreens unterbringen können und die eine gewisse Funktionalität besitzen. Ein App Widget kann eine definierte Größe haben, wobei App Widgets in Zeilen und Spalten gemessen werden, da die Homescreens ein definiertes Raster bilden, in dem App Widgets organisiert werden können.

App Widgets können drei Funktionen haben:

1. Anzeige wechselnder, aktueller Inhalte
2. Auslösen von Funktionen über Buttons und Texteingaben
3. Anzeige von Daten in Listen, Stapelansichten, ImageViews

Ein Merkmal der App Widgets ist, dass sie bei Bedarf in periodischen Abständen aktualisiert werden können. Dadurch kann z.B. der MediaPlayer als App Widget den aktuellen Titel und die aktuelle Spielzeit anzeigen oder ein Newsreader die aktuellen Nachrichten.

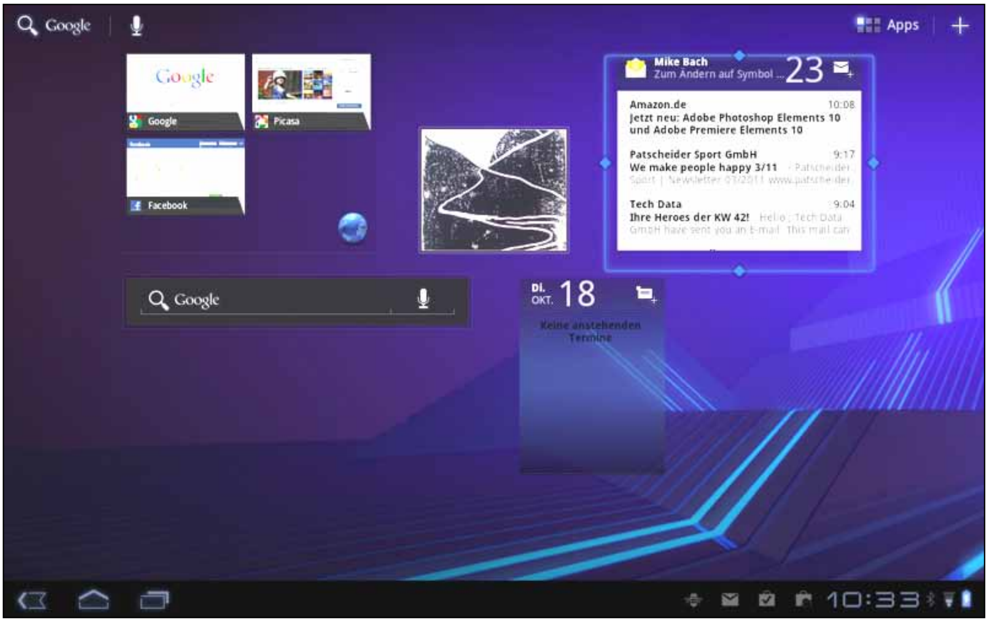


Abbildung 4.6: **Typische App Widgets auf einem Homescreen**

Das App Widget legt sich nicht über andere Anwendungen, sondern stanzt quasi ein Loch in den Homescreen und gibt den Blick auf Funktionalitäten unserer Anwendung frei. Der Homescreen ist hier der Host für das App Widget.

Wir wollen hier ein App Widget entwerfen, das die Seiten des ScrapBooks als Miniaturen anzeigen kann.

App Widgets sind im Grunde einfach Broadcast Receiver, und als solche werden die App Widgets auch im Manifest deklariert. Die Basisklasse für unsere App Widgets ist der App-WigetProvider, der ein spezieller Broadcast Receiver ist und bereits die Basisfunktionalität der App Widgets implementiert:

Listing 4.35: **ScrapBookAppWidget**

```
public class ScrapBookAppWidget extends AppWidgetProvider {

public void onUpdate(Context context, AppWidgetManager appWidgetManager,
int[] appWidgetIds) {
    final int N = appWidgetIds.length;
    File filesDir = context.getFilesDir(); //Environment.getExternalStorage➥
    PublicDirectory(Environment.DIRECTORY_PICTURES+"/ScrapBook"); //
    FilenameFilter fileNameFiler = new FilenameFilter()
    {
    public boolean accept(File dir, String filename)
    {
    return /*filename.startsWith("thumbnail_") && */filename.endsWith(".
jpg");
    }
    };
    File[] files = filesDir.listFiles(fileNameFiler);
    // Perform this loop procedure for each App Widget that belongs to this
provider
    for (int i=0; i<N; i++) {
        int appWidgetId = appWidgetIds[i];

        Intent intent = new Intent(context, ScrapBook3Main.class);
        PendingIntent pendingIntent = PendingIntent.getActivity(context, 0,
intent, 0);

        RemoteViews views = new RemoteViews(context.getPackageName(),
R.layout.scrapbook_appwidget);
        views.setOnClickPendingIntent(R.id.scrapbook_appwidget_imageview,
pendingIntent);

        if (files.length>0)
        {
        Uri uri = Uri.fromFile(files[i<files.length?i:0]);
        String sFile = uri.toString();
        views.setImageViewUri(R.id.scrapbook_appwidget_imageview, uri);
        }

        appWidgetManager.updateAppWidget(appWidgetId, views);
        }
    }

}
```

Die Methode onUpdate(...) des AppWidgetProvider wird immer dann aufgerufen, wenn das App Widget sich aktualisieren soll. Das ist:

1. wenn das AppWidget erstellt oder erstmals angezeigt wird und
2. wenn ein Aktualisierungsintervall eingestellt ist.

An einem Provider können beliebig viele Widgets hängen, die über eine ID referenziert werden. In onUpdate(...) müssen wir also dafür sorgen, alle Widgets zu durchlaufen und entsprechend zu aktualisieren.

INFO

Die Widgets selbst deklarieren wir zwar in einem Layout, das zu unserem Projekt gehört. Allerdings residieren die Widgets als RemoteViews im Host der App Widgets, also der Homescreens. Das bedeutet, wir können nicht jedes beliebige Widget innerhalb der App Widgets benutzen.

Die Verbindung zwischen unserem Provider und den RemoteViews übernehmen die Klassen `AppWidgetManager` und `RemoteViews`.

Die Klasse `RemoteViews` liefert einige Methoden, um in den in der RemoteView gekapselten »echten« Views Methoden aufzurufen. Die Klasse `RemoteViews` ist sozusagen eine »Brücke« zu den entfernten Instanzen der Views.

Unser ScrapBookAppWidget muss nun im Manifest deklariert werden. Das `android:label`-Attribut bestimmt den Namen, der auf der Widgets-Seite zum Hinzufügen von Widgets zum Homescreen angezeigt wird. Ist das Label nicht definiert, wird das Label der Applikation verwendet, ähnlich verhält es sich mit dem `android:icon`-Attribut.

Listing 4.36: **Deklaration des ScrapBookAppWidget**

```
<receiver
android:label="@string/app_name"
android:name=".appwidgets.ScrapBookAppWidget">
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
    </intent-filter>
    <meta-data android:name="android.appwidget.provider"
     android:resource="@xml/scrapbook_appwidget_info"/>
</receiver>
```

Ein App Widget benötigt noch spezielle Konfigurationsdaten, die nicht im Manifest, sondern in einer eigenen XML-Datei untergebracht werden:

```
<meta-data android:name="android.appwidget.provider"
 android:resource="@xml/scrapbook_appwidget_info"/>
```

Die Konfiguration befindet sich in einer separaten XML-Datei (hier: scrapbook_appwidget_info.xml), die per Konvention in das Verzeichnis `res/xml` gespeichert wird.

Listing 4.37: **scrapbook_appwidget_info.xml**

```
<?xml version="1.0" encoding="utf-8"?>
    <appwidget-provider xmlns:android="http://schemas.android.com/apk/res/
android"
    android:minWidth="294dp"
    android:minHeight="220dp"
    android:updatePeriodMillis="86400000"
    android:previewImage="@drawable/icon"
    android:initialLayout="@layout/scrapbook_appwidget"
    >
</appwidget-provider>
```

android:minWidth	Mindestbreite
android:minHeigh	Mindesthöhe
android:updatePeriodMillis	Aktualisierungsintervall
android:previewImage	Vorschaubild, das darstellt, wie das Widget aussehen würde, wenn es denn zugefügt und konfiguriert wurde.
android:initialLayout	Initiales Layout, das angezeigt wird, wenn das Widget noch nicht aktualisiert wird. Hier könnte man z.B. einen Text mit »Keine Daten« oder Ähnlichem anzeigen.
androisd:configure	Klassenname der Activity, die zum Konfigurieren des Widgets aufgerufen werden soll.
android:resizeMode horizontal|vertical	Ab Android 3.1 können Widgets auch interaktiv vergrößert und verkleinert werden.

Tabelle 4.6: **Attribute der AppWidgetProviderInfo**

Die Formel zur Berechnung der Breite und Höhe lautet: (Anzahl der Zellen * 74) – 2. Unser Widget ist für die Darstellung von 4 x 3 Zellen konfiguriert. Eine Zelle entspricht demnach etwa 74 dp (ca. 1,1 cm).

Vorsicht ist bei hohen Aktualisierungsraten geboten, denn zur Aktualisierung der Widgets wird das Gerät bei Bedarf auch geweckt, und das kann dann ein echter Energiefresser werden. Für Aktualisierungsintervalle z.B. im Minutenbereich wäre es besser, den `AlarmManager` zu nutzen, mit dem Intents auf Halde gelegt und nach Ablauf einer bestimmten Zeit verschickt werden.

Die interessanten Aufrufe nun in unserem App Widget sind:

```
RemoteViews views = new RemoteViews(context.getPackageName(), R.layout.
scrapbook_appwidget);
```

Hier erzeugen wir eine neue RemoteViews-Hierarchie. Das ist eine leichtgewichtige Brücke, die die eigentlichen Views umhüllt und »hostfähig« macht.

```
views.setOnClickPendingIntent(R.id.scrapbook_appwidget_imageview, pendingIn-
tent);
```

Die Methode `setOnClickPendingIntent(…)` ist eine der Brückenmethoden, um einer View (z.B. einem Button, hier der ImageView) mitzuteilen, welches Intent bei einem Klick auf das Widget ausgelöst werden soll.

```
Uri uri = Uri.fromFile(files[i<files.length?i:0]);
String sFile = uri.toString();
views.setImageViewUri(R.id.scrapbook_appwidget_imageview, uri);
```

Auch setImageViewUri(...) ist eine Brückenmethode, hier zu unserer ImageView. Wir setzen hier den URI aus dem Dateinamen eines unserer Thumbnails, die in unserem privaten Applikationsdatenverzeichnis liegen.

INFO

Die Thumbnails sind mit dem Modus MODE_WORLD_READABLE erzeugt worden! Das ist wichtig, da sonst der Hostprozess, der unser App Widget hostet, keinen Zugriff auf die Datei hat.

```
appWidgetManager.updateAppWidget(appWidgetId, views);
```

Hiermit aktualisieren wir schließlich das App Widget mit der neuen bzw. aktualisierten View.

Das Ergebnis unseres App Widgets kann sich dann schon sehen lassen:

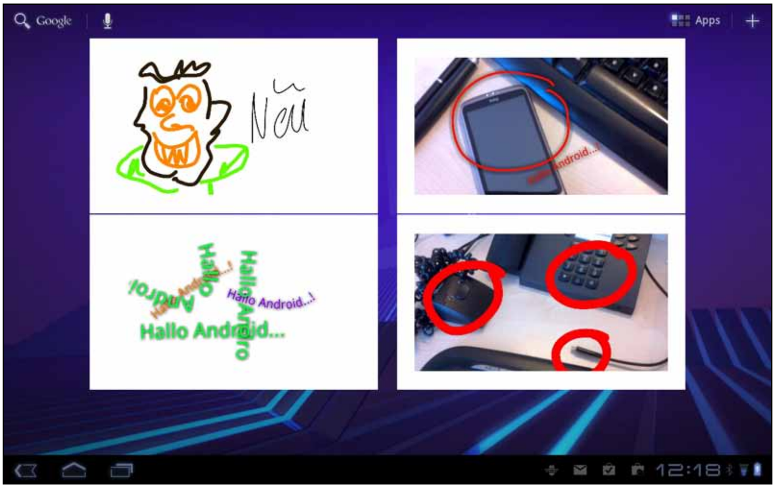


Abbildung 4.7: **Vier ScrapBook-Widgets, die jeweils eine Vorschau anzeigen**

Es können zurzeit nicht beliebige Widgets in den App Widgets-Layouts verwendet werden. Es gibt nur eine gewisse Auswahl von Layouts und Widgets, die mittels RemoteViews benutzt werden können, und Ableitungen dieser Klassen werden **nicht** unterstützt. Folgende Layouts und Widgets können verwendet werden:

» FrameLayout
» LinearLayout
» RelativeLayout
» AnalogClock
» Button
» Chronometer
» ImageButton
» ImageView
» ProgressBar
» TextView
» ViewFlipper

ACHTUNG

Man könnte jetzt auf die Idee kommen, z.B. in das FrameLayout beliebige Unterelemente einzuhängen. Das wird aber nicht funktionieren, es werden tatsächlich nur alle erdenklichen Kombinationen der obigen Layouts und Widgets unterstützt.

Das heißt für uns, dass wir komplexe Darstellungen nicht per eigenem Widget zeichnen können, sondern ggf. eine bildliche Darstellung (als PNG oder JPEG) erzeugen müssen und diese dann z.B. in eine ImageView einsetzen.

ACHTUNG

Und nochmals Achtung: Das Erstellen eines Widgets darf nicht länger als 20 Sekunden in Anspruch nehmen. Kehrt der Provider nicht innerhalb von 20 Sekunden aus dem onUpdate(...)-Aufruf zurück, erhalten wir eine »Application Not Responding«-Meldung.

Wenn wir länger brauchen sollten, müssen wir das Erstellen in einen eigenen Hintergrundservice auslagern.

Android 3 führt neben den oben genannten Layouts und Widgets noch die Möglichkeit ein, App Widgets mit Collections, also Sammlungen von Daten, zu verknüpfen, und schafft damit weitere Möglichkeiten zur Gestaltung und Funktionalität von App Widgets, die in früheren Versionen zum Teil von den Geräteherstellern durch Erweiterungen realisiert wurden. So bietet die HTC-Sense-Oberfläche schon länger App Widgets an, mit denen man durch E-Mail-Einträge scrollen kann. Das ist mit den Bordmitteln von Android vor Version 3 nicht so ohne Weiteres möglich gewesen.

Ab Version 3 können wir folgende Widgets noch hinzuziehen:

» ListView
» GridView
» StackView
» AdapterViewFlipper

Allerdings ist dazu noch ein weiterer Unterbau nötig, und zwar müssen wir einen `RemoteViewsService` und eine `RemoteViewsFactory` implementieren und bereitstellen sowie zwei Layouts bereitstellen: ein Layout für das Widget selbst, das dann eines der obigen Widgets beinhaltet, und ein Layout, das genau einen Eintrag innerhalb der Collection repräsentieren soll.

layout/scrapbook_stackappwidget.xml	Layout für das App Widget mit einer StackView.
layout/ scrapbook_stackappwidget_item.xml	Layout für **einen** Eintrag innerhalb der StackView

Tabelle 4.7: **Layouts für das App Widget**

Der RemoteViewsService stellt nun für den AppWidgetProvider die Schnittstelle zum Erstellen der RemoteViewsFactory zur Verfügung. Die Factory ist tatsächlich die Fabrik, die zum einen die zugrunde liegende Datensammlung verwaltet und zum anderen die Views/Widgets für die einzelnen Einträge erzeugt.

Die Deklaration im Manifest muss noch um diesen Service erweitert werden:

Listing 4.38: **Deklaration des RemoteViewsService**

```
<receiver android:label="ScrapBook Stack" android:name=".appwidgets.Scrap➥
    BookStackAppWidget" >
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
    <meta-data android:name="android.appwidget.provider"
     android:resource="@xml/scrapbook_stackappwidget_info" />
</receiver>

<service android:name=".appwidgets.ScrapBookStackWidgetService"
android:permission="android.permission.BIND_REMOTEVIEWS"
android:exported="false" />
```

Die Deklaration der AppWidgetProviderInfo verändert sich hingegen kaum:

Listing 4.39: **Deklaration der AppWidgetProviderInfo**

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/andro➥
    id"
    android:minWidth="294dp"
    android:minHeight="220dp"
    android:updatePeriodMillis="3600000"
    android:previewImage="@drawable/icon"
    android:initialLayout="@layout/scrapbook_stackappwidget"
    android:autoAdvanceViewId="@id/scrapbook_appwidget_stackview"
    >
</appwidget-provider>
```

Im Zusammenhang mit den Collections bzw. der StackView ist das Attribut `android:-autoAdvanceViewId="@id/scrapbook_appwidget_stackview"` interessant. Hier teilen wir dem App Widget die ID einer View mit, die automatisch weitergeschoben werden soll, was meint, dass automatisch durch die Einträge geblättert wird. Das ist besonders bei Stack-Views und dem AdapterViewFlipper interessant.

Der Service nun ist relativ einfach:

Listing 4.40: **Der RemoteViewsService**

```
public class ScrapBookStackWidgetService extends RemoteViewsService {
@Override
public RemoteViewsFactory onGetViewFactory(Intent intent) {
    return new ScrapBookRemoteViewsFactory(this.getApplicationContext(),
intent);
}
}
```

Die Basisklasse RemoteViewsService stellt bereits den kompletten Unterbau des Service bereit, wir müssen lediglich die Methode `onGetViewFactory(…)` implementieren, in der wir unsere RemoteViewFactory zurückliefern.

Diese RemoteViewFactory nun beinhaltet die eigentliche Funktionalität zum Verwalten der Daten und zum Erstellen der Views.

INFO

Intern fällt das Ganze wieder auf Adapter zurück, die Factory dient hier dazu, die entsprechenden Informationen an die Adapter, die im Host verwendet werden, zu liefern, unter anderem auch die stabilen IDs für die Einträge, die wir bei Adaptern verwenden sollten.

```
class ScrapBookRemoteViewsFactory implements RemoteViewsService.Remote-
ViewsFactory {

private Context context;
private int appId;
```

Das File[]-Array wird mit unseren Thumbnails gefüllt.

```
private File[] scrapBookFiles;

public ScrapBookRemoteViewsFactory(Context applicationContext, Intent in➥
    tent) {
    context = applicationContext;
    appId = intent.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
    AppWidgetManager.INVALID_APPWIDGET_ID);

    File filesDir = context.getFilesDir();
    FilenameFilter fileNameFiler = new FilenameFilter()
    {
    public boolean accept(File dir, String filename)
    {
        return filename.endsWith(".jpg");
    }
```

```
    };
    scrapBookFiles = filesDir.listFiles(fileNameFiler);
}

@Override
```

Hier liefern wir die Anzahl der Einträge in der Datensammlung zurück.

```
public int getCount() {
return scrapBookFiles!=null?scrapBookFiles.length:0;
}
```

Hier liefern wir die stabile (unveränderliche) ID des Eintrags zurück. In diesem einfachen Fall ist das die Position in unserem File[]-Array, bei Daten von einem Content Provider die entsprechende Datenbank-ID.

```
@Override
public long getItemId(int position) {
return position;
}
```

Hier können wir eine View erzeugen, die zwischen den Blättervorgängen angezeigt wird, falls das Laden der nächsten View länger dauern könnte.

```
@Override
public RemoteViews getLoadingView() {
return null;
}
```

Hier erstellen wir die View für den ausgewählten Eintrag.

```
@Override
public RemoteViews getViewAt(int position) {
```

Der folgende Aufruf konstruiert eine RemoteView-Hierarchie, die das Layout für den **Eintrag** benutzt.

```
RemoteViews rv = new RemoteViews(context.getPackageName(), R.layout.scrap➥
    book_stackwidget_item);
```

Unsere ImageView erhält den URI, der auf das Thumbnail zeigt.

```
Uri uri = Uri.fromFile(scrapBookFiles[position]);
rv.setImageViewUri(R.id.scrapbook_appwidget_imageview, uri);

return rv;
}
```

Jeder Eintrag könnte potenziell, abhängig von irgendeinem Kriterium, anders aussehen und somit einen anderen Typ von View für den Eintrag liefern. Hier gäbe man die Anzahl der unterschiedlichen Darstellungsarten zurück.

```
@Override
public int getViewTypeCount() {
return 1;
}
```

Teilt dem Framework mit, ob die IDs der Datensammlung unveränderlich sind. Unveränderliche IDs erlauben einige Optimierungen, sind also immer ratsam.

```
@Override
public boolean hasStableIds() {
return true;
}
```

Die Methode `onCreate()` wird aufgerufen, wenn die Factory das erste Mal konstruiert und benötigt wird.

```
@Override
public void onCreate() {
}
```

Wird aufgerufen, falls sich die zugrunde liegenden Daten verändert haben sollten. Die Datenänderung muss die Anwendung, die die Daten verwaltet, mittels `AppWidgetManager.notifyAppWidgetViewDataChanged(…)` anzeigen.

```
@Override
public void onDataSetChanged() {
}
```

Hier müssen wir alles aufräumen, was wir im Zusammenhang mit dieser Datensammlung an Ressourcen verbraucht haben, z.B. Cursor schließen, Bitmaps recyceln oder Ähnliches.

Listing 4.41: **Die Factory**

```
@Override
public void onDestroy() {
scrapBookFiles = null;
}

}
```

Innerhalb des AppWidgetProviders nun wird der Service als Remoteadapter an unser App Widget gebunden:

Listing 4.42: **Binden des RemoteViewsService an unser App Widget**

```
public class ScrapBookStackAppWidget extends AppWidgetProvider {
public static final String SCRAPBOOK_ACTION = "de.androidpraxis.scrapbook3.
SCRAPBOOK_ACTION";
public static final String EXTRA_ITEM = "de.androidpraxis.scrapbook3.EXT➥
    RA_ITEM";

@Override
public void onUpdate(Context context, AppWidgetManager appWidgetManager,
int[] appWidgetIds) {

for (int i = 0; i < appWidgetIds.length; ++i) {

    Intent intent = new Intent(context, ScrapBookStackWidgetService.class);
    intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetIds[i]);
    intent.setData(Uri.parse(intent.toUri(Intent.URI_INTENT_SCHEME)));

    RemoteViews rv = new RemoteViews(context.getPackageName(), R.layout.
scrapbook_stackappwidget);
    rv.setRemoteAdapter(appWidgetIds[i], R.id.scrapbook_appwidget_stackview,
intent);

    rv.setEmptyView(R.id.scrapbook_appwidget_stackview, R.id.empty_view);
    appWidgetManager.updateAppWidget(appWidgetIds[i], rv);
}
super.onUpdate(context, appWidgetManager, appWidgetIds);
}
}
```

Das Entscheidende hier ist, dass der RemoteViewsService per `setRemoteAdapter(…)` an die Hauptview unseres Widgets gebunden wird. Dabei nutzt das System ein Intent, das mit dem Service initialisiert wird:

```
Intent intent = new Intent(context, ScrapBookStackWidgetService.class);
intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetIds[i]);
intent.setData(Uri.parse(intent.toUri(Intent.URI_INTENT_SCHEME)));

RemoteViews rv = new RemoteViews(context.getPackageName(), R.layout.scrap➥
    book_stackappwidget);
rv.setRemoteAdapter(appWidgetIds[i], R.id.scrapbook_appwidget_stackview,
intent);
```

INFO

Das `setData(…)` auf den URI des Intents ist hier wichtig, damit das Intent gegen andere Intents verglichen werden kann. Beim Vergleich von Intents werden nämlich die Einträge mit `putExtra(…)` nicht berücksichtigt. Eindeutig wird das Intent hier durch das `setData(…)`.

INFO

Das ist wieder ein schönes Beispiel dafür, wie die Kommunikation über Prozessgrenzen über die Intents abgehandelt werden kann. Tief im System sind zwar die RemoteViews selbst als IPC-Objekte (Inter Process Communication) mit Remote Procedure Calls bzw. Remote Method Invocation realisiert, der Transport bestimmter Objekte zwischen den Prozessen wird aber über den Standardmechanismus des Intents erledigt, der ja Grundlage für die Kopplung der Anwendungsmodule ist.

Auf diese Weise haben wir unserem ScrapBook ein weiteres App Widget spendiert, mit dem der Anwender durch die Vorschaudarstellung der ScrapBook-Seiten auf dem Homescreen blättern kann.

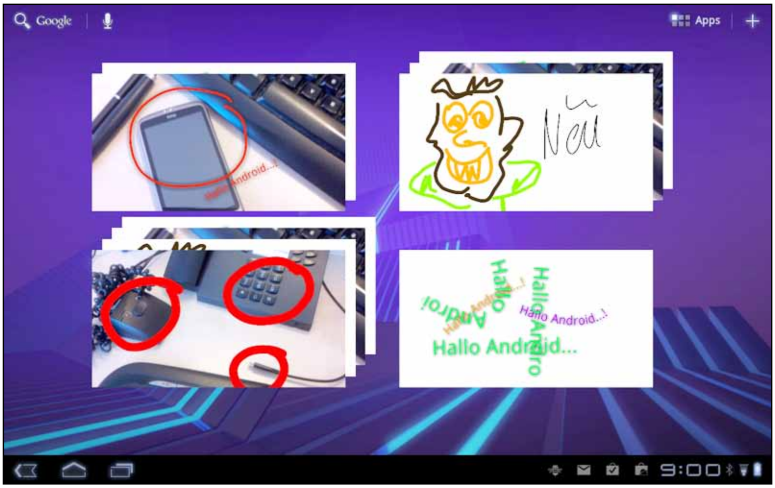


Abbildung 4.8: **App Widget mit StackWidget**

4.4 Sensoren

Die verschiedenen möglichen Sensoren haben wir im ersten Kapitel bereits kennengelernt. Wie wir die Sensoren nutzen können, wollen wir uns hier anschauen. Dazu wenden wir uns wieder dem MarbleGame zu, das die Sensoren dazu benutzt, die Kugel in die Richtung der Geräteneigung zu beschleunigen. Da das MarbleGame aus einer Simulation physikalischer Vorgänge entstanden ist, wollen wir den Anteil der »Gravitationskraft«, der durch die Neigung auf die Kugel wirkt, entsprechend berücksichtigen.

Um Sensorwerte zu empfangen, müssen wir uns beim `SensorManager` als `SensorEventListener` registrieren. Wir fragen also die Werte nicht ab, sondern horchen darauf, dass uns Werte übergeben werden.

Wir entscheiden, abhängig von der Anforderung, welchen Sensortyp wir abhören wollen, und registrieren uns für diesen Sensortyp. Wir können uns für einen oder auch mehrere Sensortypen registrieren.

ACHTUNG

Die Übermittlung von Sensorwerten verbraucht Energie. Es ist wichtig, dass wir uns vom `SensorManager` **abmelden, wenn wir die Sensorwerte nicht (mehr) benötigen.**

```
private void initSensorManagement() {
    this.sensorManager = (SensorManager)getSystemService(SENSOR_SERVICE);
```

Wir lassen uns alle verfügbaren Sensoren geben.

```
    List<Sensor> sensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
    Iterator<Sensor> iter = sensors.iterator();

    gameAccelerometerSensor = null;
    gameMagneticFieldSensor = null;

    while (iter.hasNext())
    {
        Sensor sensor = iter.next();
        if (sensor.getType() == Sensor.TYPE_ACCELEROMETER)
        {
        if (this.gameAccelerometerSensor == null)
        {
```

Der Beschleunigungssensor ist der Sensor, der uns für die Steuerung interessiert.

```
        this.gameAccelerometerSensor = sensor;
        }
        }
        if (sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD)
        {
        if (this.gameMagneticFieldSensor == null)
        {
```

Den Sensor für das magnetische Feld nehmen wir auch mit.

Listing 4.43: **Heraussuchen der Sensoren**

```
        this.gameMagneticFieldSensor = sensor;
        }
    }
    Log.d(Globals.LOG_TAG,this.getClass().getName()+".initSensorManage➥
    ment(): "+sensor.getName()+" "+SensorHelper.sensorTypeName(sensor.getTy➥
    pe()));
    }

}
```

Der `SensorManager` ist der Dreh- und Angelpunkt für die Nutzung der Sensoren. Im obigen Listing sehen wir, wie wir die verfügbaren Sensoren aufzählen und uns die Sensoren merken, die wir abhören wollen.

Man kann auch direkt den gewünschten Typ aufzählen lassen, hier können wir aber (in der Log-Ausgabe) auch sehen, welche Sensoren unser Gerät überhaupt besitzt.

Damit wir nun die Sensorwerte empfangen können, registrieren wir die Game-Activity als SensorEventListener:

```
public class Game extends Activity implements SensorEventListener {
[…]
@Override
protected void onResume()
{
```

Die Methode `onResume()` wird immer aufgerufen, bevor die Activity wieder mit dem Anwender interagiert. Das ist auch der späteste Zeitpunkt, zu dem wir Sensorwerte empfangen wollen. Wir realisieren das an dieser Stelle, weil wir, wie wir im Folgenden sehen, uns von den Sensoren beim Pausieren wieder abmelden müssen.

```
super.onResume();
    Log.d(Globals.LOG_TAG,this.getClass().getName()+".onResume()");
    if (gameAccelerometerSensor != null) this.sensorManager.
registerListener(this, gameAccelerometerSensor, SensorManager.SENSOR_DELAY_
GAME);
    if (gameMagneticFieldSensor != null) this.sensorManager.
registerListener(this, gameMagneticFieldSensor, SensorManager.SENSOR_DELAY_
GAME);
    playgroundView.resumeGame();
}

@Override
protected void onPause()
{
    super.onPause();
    Log.d(Globals.LOG_TAG,this.getClass().getName()+".onPause()");
```

Das ist sehr wichtig, um Batterie zu sparen. Wenn die Anwendung pausiert, werden ansonsten die Sensorwerte trotzdem weiter an die Anwendung geschickt.

Listing 4.44: **Registrieren der Game-Activity**

```
    sensorManager.unregisterListener(this);

    playgroundView.pauseGame();
}
}
```

Mittels `this.sensorManager.registerListener(this, gameAccelerometerSensor, SensorManager.SENSOR_DELAY_GAME)` registrieren wir uns und geben an, mit welcher zeitlichen Auflösung die Werte an uns übermittelt werden sollen:

SensorManager.SENSOR_DELAY_FASTEST	So schnell wie möglich
SensorManager.SENSOR_DELAY_GAME	Für Spielesteuerung (so schnell wie nötig)
SensorManager.SENSOR_DELAY_NORMAL	Normale Geschwindigkeit, z.B. um Änderung der Bildschirmlage zu erkennen
SensorManager.SENSOR_DELAY_UI	Nicht ganz so schnell, brauchbar, wenn die Werte in einer »normalen« Anwendung benötigt werden.

Tabelle 4.8: **Übermittlungsraten für Sensorwerte**

INFO

Je schneller die Übermittlung, umso mehr Energie wird verbraucht. Auch ist es nicht garantiert, dass ein Sensor seine Werte kontinuierlich übermittelt. Bis Android 2.2 wurden immer nur die Wertänderungen übergeben, ab Version 2.3 können Sensorwerte auch kontinuierliche Datenströme abliefern.

Ist die Activity erst einmal als SensorEventListener registriert, können wir die Werte abfangen und verarbeiten:

```
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
```

Hier können wir darauf reagieren, wenn die Genauigkeit der Datenlieferung sich ändert. Sensoren geben damit Auskunft, wie verlässlich die Daten sind oder ob ggf. eine Kalibrierung o.Ä. nötig ist. Die möglichen Werte sind SensorManager.SENSOR_STATUS_ACCURACY_HIGH, SensorManager.SENSOR_STATUS_ACCURACY_MEDUIM und SensorManager.SENSOR_STATUS_ACCURACY_LOW.

```
}

@Override
public void onSensorChanged(SensorEvent event) {
```

Diese Methode wird durch den SensorManager aufgerufen, wenn (neue) Werte am Sensor anliegen. Die Übermittlung der Werte und eines Zeitstempels sowie des Genauigkeitsindikators erfolgt über das SensorEvent-Objekt.

```
    float[] event_values = event.values.clone();
```

Es ist wichtig, die Werte aus dem Event zu klonen, wenn wir diese weitergeben und irgendwo speichern. Die SensorEvent-Strukturen, insbesondere die Werte, sind in der Regel so angelegt, dass es sich um gemeinsam genutzten Speicher handelt, in dem der Sensor z.B. immer die aktuellen Werte ablegt. Wenn wir also nur die Referenz weitergeben, dann finden wir nicht unbedingt die Werte vor, die übermittelt wurden.

```
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
    {
```

Hier übergeben wir die Werte jeweils an einen weiteren Empfänger, abhängig vom Sensortyp.

Listing 4.45: **Empfangen der Sensorwerte im SensorEventListener**

```
        playgroundView.getEnvironment().setGravityVector(event_values);
    }
    if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD)
    {
        playgroundView.getEnvironment().setGeomagneticVector(event_values);
    }

}
```

Das SensorEvent besitzt folgende Eigenschaften:

int accuracy	Genauigkeit der Werte: SensorManager.SENSOR_STATUS_ACCURACY_HIGH SensorManager.SENSOR_STATUS_ACCURACY_MEDUIM SensorManager.SENSOR_STATUS_ACCURACY_LOW.
Sensor sensor	Der Sensor, der das Event ausgelöst hat
long timestamp	Zeitstempel in Nanosekunden, zu dem das Event ausgelöst wurde.
float values[]	Die Werte, die der Sensor übermittelt. Es hängt vom Typ des Sensors ab, wie viele Werte im Array übergeben werden. Viele Sensoren sind Dreiachser und übermitteln drei Werte, der Lichtsensor z.B. nur einen Wert.

Tabelle 4.9: **Eigenschaften des Sensor-Events**

INFO

Die Dreiachser liefern die Werte in der Reihenfolge X,Y,Z im Sensorkoordinatensystem, bei dem die Z-Achse aus dem Display auf uns zu, die X-Achse nach rechts und die Y-Achse nach oben zeigt. Eine Diskussion über die Koordinatensysteme findet sich in Kapitel 1. Es gibt Situationen, in denen man die Lage des Geräts berücksichtigen muss, z.B. bei einer Kompassanwendung, wenn das Gerät wie eine Kamera aufrecht gehalten wird oder flach (Display schaut zum Himmel) wie ein Kompass.

In der Spielwiese ist eine Activity (Sensor) implementiert, die alle verfügbaren Sensoren auflistet, ein paar Sensoren ableitet und die Werte in einem X/Y-Schreiber bzw. als Winkel darstellt. Das ist ein guter Ausgangspunkt für eigene Experimente mit den Sensoren und um die Änderung der Werte zu verstehen, wenn das Gerät z.B. geneigt wird.

Im MarbleGame übergeben wir die Werte des Acceleration-Sensors (Beschleunigungssensor) als Gravity-Vektor (Vektor mit den Anteilen der Erdbeschleunigung). Sachlich nicht ganz korrekt, aber da definitionsgemäß der Beschleunigungssensor die Erdbeschleunigung **nicht** herausrechnet, beinhaltet der Vektor den Einfluss der Erdbeschleunigung auf die Achsen. Die Definition lautet folgendermaßen:

values[0]:	Beschleunigung des Geräts minus Gx (Gravitation in X-Richtung) in X-Richtung
values[1]:	Beschleunigung des Geräts minus Gy (Gravitation in Y-Richtung) in Y-Richtung
values[2]:	Beschleunigung des Geräts minus Gz (Gravitation in Z-Richtung) in Z-Richtung

Tabelle 4.10: **Bedeutung der Werte des Beschleunigungssensors**

Das heißt: Liegt das Gerät in Ruhe auf dem Tisch, dann haben wir in Z-Richtung eine Beschleunigung von 9,81 m/s^2.

Halten wir das Gerät wie eine Kamera vor uns, dann ist der Anteil der Erdbeschleunigung in der Z-Richtung nahe 0 m/s^2, dafür in X-Richtung bzw. Y-Richtung nahe +/– 9,81 m/s^2, je nachdem, ob wir das Gerät hochkant oder quer vor uns halten.

Es ist sehr schön zu sehen, dass mit dem Accelerometer bereits eine Abschätzung der Lage im Raum getroffen werden kann, und üblicherweise arbeitet die Erkennung für die Bildschirmausrichtung damit. Das erklärt auch, warum der Bildschirm, wenn das Gerät flach auf dem Tisch liegt, nicht mehr rotieren kann: Die Beschleunigung auf die X-Achse und die Y-Achse ist ja dann nahe 0 m/s^2.

Ein Sensor, der uns einen Kompass liefert, kann also nicht alleine durch den Beschleunigungssensor realisiert werden, es muss noch ein Sensor her, der die Lage des Geräts im Erdmagnetfeld berücksichtigt. Erst dann haben wir genug Komponenten, um die Himmelsrichtung (mehr oder weniger genau) zu bestimmen.

Im MarbleFame benutzen wir den Vektor, um die Beschleunigung der Kugel in X-Richtung zu bestimmen.

ACHTUNG

Hier müssen wir dann aufpassen. Das MarbleGame wird im Querformat betrieben. Das heißt, dass die X-Achse des Bildschirms in negativer Richtung der Y-Achse des Sensorkoordinatensystems zeigt, wie die unten stehende Abbildung verdeutlicht, falls das Gerät eigentlich ein Hochkantgerät ist.



Abbildung 4.9: **Koordinatensystem des MarbleGame-Bildschirms**

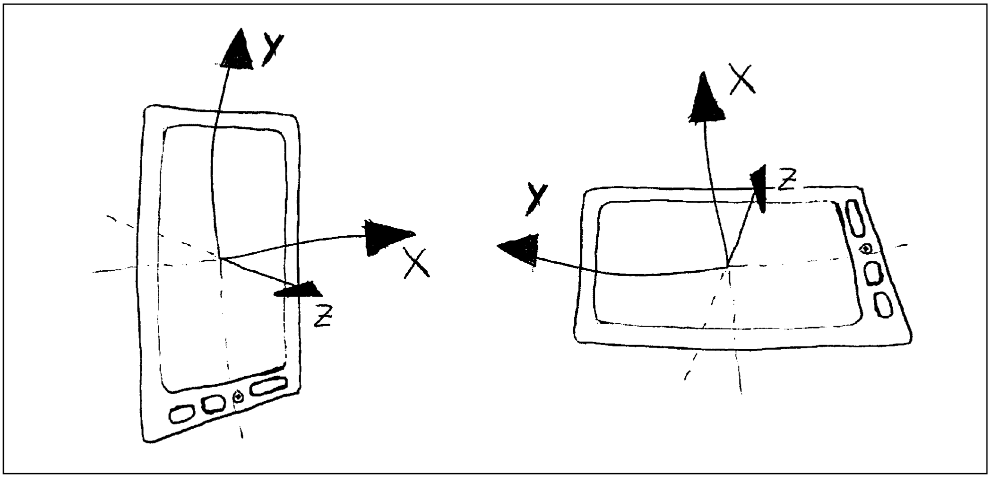


Abbildung 4.10: **Koordinatensystem des Sensors, links noch aufrecht, rechts wenn das Gerät gekippt wurde**

Wenn wir also die Werte des Vektors verarbeiten, dann müssen wir uns über die Ausrichtung im Klaren sein. Wir könnten nun hergehen und die Verarbeitung der übergebenen Werte fest verdrahten, also immer die Y-Komponente nehmen, wenn wir sagen, das Gerät wird immer im Querformat betrieben, **aber**: Die natürliche Ausrichtung des Geräts spielt hierbei eine große Rolle. Tablets sind z.B. meist bereits im Querformat, und dann ist das Sensorkoordinatensystem ebenfalls so ausgerichtet. In diesem Fall zeigt nicht die Sensor-Y-Koordinate in die X-Richtung des Bildschirms, sondern tatsächlich die X-Koordinate.

Um das zu berücksichtigen und, als »Abfallprodukt«, das Spiel auch so konzipieren zu können, dass es auch hochkant und über Kopf funktioniert, müssen wir die Orientierung prüfen:

Listing 4.46: **Prüfen der Ausrichtung und Bestimmen der Sensorkoordinate**

```
public void checkOrientation() {
    int rotation = ((WindowManager)context.getSystemService(Context.WINDOW_
SERVICE)).getDefaultDisplay().getRotation();
    if (rotation==Surface.ROTATION_0 || rotation ==Surface.ROTATION_180)
    {
    //Keine Rotation gegenüber der natürlichen Lage -> Sensorkoordinaten
stimmen mit Bildschirm überein
        accelerationYaxisIndex = 1;
        accelerationXaxisIndex = 0;

    if (rotation ==Surface.ROTATION_0)
    {
        accelerationDirection = -1;
    }
    if (rotation ==Surface.ROTATION_180)
    {
        accelerationDirection = 1;
    }
    }
    else
    {
```

```
    //Rotation gegenüber der natürlichen Lage -> Sensorkoordinaten sind um 
90° gedreht
        accelerationYaxisIndex = 0;
        accelerationXaxisIndex = 1;
    if (rotation ==Surface.ROTATION_90)
    {
        accelerationDirection = 1;
    }
    if (rotation ==Surface.ROTATION_270)
    {
        accelerationDirection = -1;
    }
    }
}
```

Nach dieser Prüfung stehen die Achse und ihre Ausrichtung gegenüber unseren Bildschirmkoordinaten fest.

Der Sensorwert wird nun folgendermaßen verarbeitet:

```
private void updatePhysics() {
    if (state == STATE_RUNNING)
    {
```

Die tick()-Methode ist die Methode, um die Simulation in unserem Simulationszeitfenster weiterticken zu lassen, hier wird dann der aktuelle Geschwindigkeitsvektor berechnet und auf Basis dieses Vektors die neue Position.

```
        theBody.tick();
```

Der `GravityVector` wurde durch den SensorEventListener gesetzt.

```
        float[] gravity = environment.getGravityVector();
```

Die Richtung und die Achse der Beschleunigung wurden bei der Prüfung der Bildschirmrotation gesetzt.

```
        float acceleration = accelerationDirection * gravity[accelerationXax
isIndex];
```

In unserer Spielumwelt gibt es eine Gravitationskonstante für die Beschleunigung in Y-Richtung, damit können wir auch auf andere Planeten ausweichen.

Listing 4.47: **Verwenden des Sensorwerts**

```
        Vector vg = new Vector( acceleration, environment.getGravityCons➥
    tant(), 0 );
        Log.d(Globals.LOG_TAG, "Acceleration X: "+acceleration);
        theBody.setAcceleration(vg);
    }
}
```

Damit haben wir den Beschleunigungssensor als Steuerungsmechanismus für unser Spiel eingesetzt. Neigen wir das Gerät nach links oder rechts, dann wird die Kugel entsprechend um den Betrag in diese Richtung beschleunigt.

TIPP

Das MarbleGame hat eine (einfache) Physik-Engine, mit der ich versucht habe, Beschleunigung, die Abprallbewegungen und auch die Dämpfung durch Elastizität und Luftwiderstand einigermaßen natürlich zu simulieren. Bedient euch einfach.

Interessant sind natürlich noch die anderen Sensoren. Mit der Spielwiese lassen sich alle Sensortypen ganz nett untersuchen. Ein wirklich interessanter Fall ist aber eine Kompassanwendung. Zwar führen die Geräte schon einen abgeleiteten Richtungssensor mit, der die Drehung um die jeweiligen Achsen in Grad übermittelt, aber empfohlen wird seit geraumer Zeit, die Himmelsrichtung über den Beschleunigungssensor und das Magnetometer (Teslameter) selbst zu ermitteln. Hier wird dann wieder die Ausrichtung des Geräts wichtig, denn es ist ein Unterschied (in den Vektoren), ob wir das Gerät flach halten oder, wie man es bei Augmented-Reality-Lösungen machen muss, als Kamera vor uns.

Normalerweise, wenn das Gerät flach auf dem Tisch liegt, zeigt die Rotation der Y-Achse um die Z-Achse der Sensoren die Himmelsrichtung an, bei 0° Drehung um die Z-Achse zeigt die Y-Achse nach Norden und die X-Achse nach Osten.

In diesem Fall lässt sich aus dem Beschleunigungsvektor und dem Gravitationsvektor einfach eine Rotationsmatrix ermitteln und daraus wiederum einen Rotationsvektor des Gerätekoordinatensystems um das Weltkoordinatensystem.

Wenn nun aber das Gerät im Kameramodus gehalten wird, dann ist die Himmelsrichtung entweder die Drehung der Z-Achse um die Y-Achse oder die Drehung der Z-Achse um die X-Achse, je nach Haltung **und** natürlicher Ausrichtung des Geräts. In so einem Fall muss das Koordinatensystem bei der Berechnung transformiert, »gemappt«, werden. Die folgende Methode stammt aus der SensorManagement-Klasse aus meiner `SystemAndHardwareLibrary`, in der ich einige Dinge rund um die Hardware und Sensoren gesammelt habe.

```
public boolean getOrientation(float[] values)
{
float[] R = new float[16];
```

Die Eigenschaften `acceleration` und `magneticfield` werden durch die Sensoren gesetzt, die Klasse `SensorManagement` fungiert hier selbst als SensorEventListener und richtet entsprechende Handler ein.

```
if (acceleration!=null && magneticfield!=null)
{
```

Mit `getRotationMatrix(…)` berechnen wir eine Rotationsmatrix, die sich aus der Orientierung des Geräts im Magnetfeld und den Werten des Beschleunigungssensors ermitteln lässt.

```
if (SensorManager.getRotationMatrix(R, null, acceleration, magneticfield))
{
[…]
```

Mit `getOrientation(…)` können dann aus der Rotationsmatrix R die Rotation der Geräteachsen und das Weltkoordinatensystem bestimmt werden. Liegt das Gerät flach auf dem Tisch, so steht danach in `values[0]` die Himmelsrichtung in Grad.

Listing 4.48: **Bestimmen der Rotation des Gerätekoordinatensystems**

```
SensorManager.getOrientation(R, values);
return true;
}
}

return false;
}
```

Wenn wir aber das Gerät nun im Kameraformat halten, dann müssen wir mittels `remapCoordinateSystem(…)` die Achsen der Rotationsmatrix vertauschen, denn dann ist ja entweder die X-Achse oder die Y-Achse die Drehachse. Das bedeutet, die Z-Achse des Geräts wird auf die Y-Achse der Welt gekippt, denn im Kameramodus wollen wir ja auch, dass die Y-Achse bei 0° Drehung um die Z-Achse nach Norden zeigt, nur dass jetzt die Z-Achse des Geräts mit der Y-Achse der Welt zur Deckung gebracht wird. In diesem Fall sind die natürliche Ausrichtung des Geräts und die Rotation unerheblich, da immer die Z-Achse zur Y-Achse wird. Hier muss man nur auf die korrekte Ausrichtung des Kamerabildes achten.

Wenn wir einen mechanischen Kompass simulieren wollen, dann müssen wir noch berücksichtigen, in welcher Ausrichtung wir das Gerät betreiben wollen, was also die »Peilseite« des Kompasses ist, über die wir die Peilung vornehmen wollen. Wollen wir in Richtung der natürlichen Ausrichtung schauen, dann muss keine Anpassung durchgeführt werden. Wenn wir aber über die andere Seite peilen wollen, also das Gerät um 90° gedreht betreiben, dann müssen die Y-Achse und die negative X-Achse vertauscht werden.

Um also die Rotationsmatrix entsprechend anzupassen, erfolgt ein Aufruf von `remapCoordinateSystem(…)` unter Berücksichtigung der Orientierung und der natürlichen Ausrichtung:

```
if (deviceMode == CAMERA_MODE)
{
```

Gerät wird im Kameramodus betrieben:

```
    SensorManager.remapCoordinateSystem(R, SensorManager.AXIS_X, SensorMana➥
    ger.AXIS_Z, R);
}
case DEVICE_COMPASS_MODE_0:
```

Peilung in Richtung der natürlichen Ausrichtung, Achsen werden identisch abgebildet.

```
    SensorManager.remapCoordinateSystem(R,
    SensorManager.AXIS_X, SensorManager.AXIS_Y, R);
    break;

case DEVICE_COMPASS_MODE_90:
```

Peilung bei Betrieb des Geräts um 90° gedreht, also entweder Hochkantgerät im Querformat oder Tablet im Hochformat, Achsen werden getauscht, die Richtung der Y-Achse invertiert.

Listing 4.49: **Betrieb im Kameramodus, um die Blickrichtung festzustellen**

```
    SensorManager
    .remapCoordinateSystem(R, SensorManager.AXIS_Y,
    SensorManager.AXIS_MINUS_X, R);
    break;
[…]
```

Das »Remapping« bedeutet, welche Achse des Geräts auf welche Achse (und welche Richtung) der Welt abgebildet wird. Kippen wir das Gerät in den Kameramodus, dann kippt die Z-Achse des Geräts auf die Y-Achse der Welt.

ACHTUNG

In einer wirklichen Wandersituation sollte man sich nicht auf die Gerätekompasse verlassen, sondern einen ordentlichen Kompass kaufen. Das hier vorgestellte Verfahren stellt keine Möglichkeiten zur Verfügung, die Genauigkeit der Werte zu überprüfen. Beide Sensoren, die in die Richtungsbestimmung einbezogen werden, liefern je nach Umgebung schwankende Werte und mitunter auch ein ziemliches Rauschen. In einer »echten« Anwendung, die eine gute Genauigkeit erfordert, müsste man das Rauschen eliminieren und auch die Schwankungen im Magnetfeld sehr genau beobachten, um die Genauigkeit zu erhöhen bzw. überhaupt beurteilen zu können.

Damit lässt sich dann in Verbindung mit der Kameravorschau und einem Overlay sehr schön die Blickrichtung in die Vorschau einblenden. Ein Beispiel dafür findet sich ebenfalls in der Spielwiese.

4.5 Location Services

Ein weiterer Sensor bzw. weitere Sensoren dienen dem Feststellen der Position des Geräts in der weiten Welt. Android bietet die Positionsbestimmung in zwei Varianten an, die sich in der Genauigkeit, aber auch in Geschwindigkeit und Energieverbrauch unterscheiden.

Zum einen kann Android die Position aus den Zellinformationen des Mobilfunks und WiFi-Hotspots ermitteln. Diese Variante benötigt wenig Energie, arbeitet auch innerhalb geschlossener Räume, ist aber auch nicht so genau.

Zum anderen besitzen alle Android-Geräte einen GPS-Sensor zur Positionsbestimmung. Mittels GPS ist eine genauere Positionsbestimmung möglich. Ohne besondere Fehlerkorrektur und unterstützende Maßnahmen wird die Genauigkeit seit 2000 zwischen 7,8 m und 15 m angegeben. Mittels Assisted GPS und weiterer Tricks soll es möglich sein, Genauigkeiten unter 7 m zu erreichen, das hängt aber sehr stark von äußeren Einflüssen ab.

Grundsätzlich können wir zwischen beiden Methoden wählen oder aber beide Methoden nutzen und das bessere Ergebnis für die Positionsbestimmung verwenden. Die Strategie bei der Positionsbestimmung ist, jeden neuen Fix, so heißt der Zeitpunkt, zu dem eine Position vorliegt, mit dem aktuellen (besten) Fix zu vergleichen und jeweils nur die bessere Variante zu wählen.

Die Güte des Fixes kann man anhand dreier Kriterien beurteilen:

Location.getAccuracy()	Liefert die Genauigkeit in Metern oder 0,0 m, wenn die Genauigkeit unbestimmt ist.
Location.getTime()	Zeitpunkt des Fixes. Je neuer, je besser.
Location.getProvider()	Liefert LocationManager.GPS_PROVIDER oder LocationManager.NETWORK_PROVIDER. GPS-Provider sind, in der Regel, genauer als der Network-Provider.

Tabelle 4.11: **Drei Kriterien zur Auswahl eines Fixes**

Um überhaupt Positionsbestimmung durchführen zu können, müssen wir die entsprechende Erlaubnis im Manifest anfordern:

<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />	GPS-Provider und Netzwerkprovider benutzen.
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />	Nur den Netzwerkprovider benutzen.

Tabelle 4.12: **Berechtigungen für die Positionsbestimmung**

Ähnlich wie bei der Nutzung der Sensoren ist der Dreh- und Angelpunkt für die Positionsbestimmung der `LocationManager`. Über den `LocationManager` können wir uns als `LocationListener` registrieren, um Änderungen der Fixes zu empfangen, und wir können den Status der Provider und sogar den Status der GPS-Satelliten auswerten.

In Verbindung mit dem ScrapBook wollen wir die Positionsbestimmung dazu nutzen, unsere Scribbles mit der aktuellen Position zu markieren (Geo-Tagging), um später in Google Maps anzeigen zu können, wo wir das Scribble erstellt haben.

INFO

Der Content Provider für Bilder beinhaltet die Spalten MediaStore.Images.ImageColumns. LATITUDE und MediaStore.Images.ImageColumns.LONGITUDE. Wenn wir die Kameraapplikation so eingestellt haben, dass sie die Position im Bild speichern soll, können wir diese Werte auch verwenden, wenn wir das Bild aus der Galerie lesen oder die Kameraanwendung in unsere Applikation einbetten.

Schauen wir uns die Schritte an, um die Position zu ermitteln:

```
public class LocationManagement implements LocationListener {

private Context context;
private LocationManager locationManager;
private Location currentBestLocation = null;

public static interface LocationFixListener
{
void betterFixAvailable(Location location);
}

private LocationFixListener locationFixListener = null;

public LocationManagement(Context context)
{
this.context = context;
```

Hier holen wir uns eine Referenz auf den LocationManager.

```
this.locationManager = (LocationManager)context.getSystemService(Context.
LOCATION_SERVICE);
}

/////////////////////////////////////////////////////////////////////////////
////////////////
```

Die folgende Methode startet das Horchen auf Positionsdaten. Der LocationFixListener ist ein Listener, den ich selber gebaut habe, um einfach aus dieser Klasse den besten Fix nach außen melden zu können. Dadurch wird die gesamte Positionsbestimmungslogik in einer wiederverwendbaren Klasse gekapselt und kann in verschiedenen Projekten eingesetzt werden.

```
public void start(boolean useGPS, LocationFixListener locationFixListener)
{
    this.locationFixListener = locationFixListener;
```

Nun starten wir das Abhören von Positionsdaten auf dem Network-Provider ...

```
    locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,
0, 0, this);
```

... und bei Bedarf noch das Abhören auf dem GPS-Provider:

```
    if (useGPS)
    {
        locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,
0, 0, this);
```

Der LocationManager bietet einen Zugriff auf die letzte bekannte Position der jeweiligen Provider. Damit können wir schon eine möglicherweise brauchbare Position benutzen, ohne auf den ersten Fix warten zu müssen.

```
        currentBestLocation = locationManager.getLastKnownLocation(Location➥
    Manager.GPS_PROVIDER);
    }

    Location tmpLocation = locationManager.getLastKnownLocation(Location-
Manager.NETWORK_PROVIDER);
```

Hier wird geschaut, ob der GPS-Provider oder der Netzwerkprovider die bessere zuletzt bekannte Position geliefert hat. Die Methode `isBetterLocation(…)` habe ich ebenfalls in dieser Klasse realisiert (und aus dem Google-Beispiel »entwendet«) und dient dazu, anhand der Genauigkeit, des Zeitstempels und des liefernden Providers den mutmaßlich besseren Fix auszuwählen.

```
    if (isBetterLocation(tmpLocation,currentBestLocation))
    {
        currentBestLocation = tmpLocation;
    }

    if (currentBestLocation!=null)
    {
    if (locationFixListener!=null)
    {
```

Hier melden wir die letzte gute Position als »Scheinfix« nach außen.

```
      locationFixListener.betterFixAvailable(currentBestLocation);
    }
    }
}
```

Hiermit werden alle Listener abgemeldet. Es ist wichtig, das Abhören in einem gewissen Zeitfenster zu beenden, mindestens jedoch dann, wenn die Anwendung schlafen gelegt wird, damit der Batterieverbrauch begrenzt wird.

```
public void stop()
{
    locationManager.removeUpdates(this);
}
```

Hier landen die Fixes von unseren LocationProvidern. Innerhalb dieser Methode wird wieder gegen den zuletzt erhaltenen Fix geprüft und der bessere von beiden nach außen gemeldet.

```
@Override
public void onLocationChanged(Location location) {

    if (locationFixListener!=null)
    {
        if (isBetterLocation(location,this.currentBestLocation))
        {
        this.currentBestLocation = location;
        locationFixListener.betterFixAvailable(location);
        }
}
}
```

Diese Methode wird aufgerufen, wenn ein Provider abgeschaltet wird. Über `locationManager.isProviderEnabled(String provider)` können wir auch vorher abfragen, ob der Provider überhaupt zur Verfügung steht.

```
@Override
public void onProviderDisabled(String location) {
}
```

Diese Methode wird aufgerufen, wenn ein Provider angeschaltet wird. Über `locationManager.isProviderEnabled(String provider)` können wir auch vorher abfragen, ob der Provider überhaupt zur Verfügung steht.

```
@Override
public void onProviderEnabled(String location) {
}
```

Diese Methode wird aufgerufen, wenn sich der Status des Providers verändert. Mögliche Statusinformationen sind OUT_OF_SERVICE TEMPORARILY_UNAVAILABLE und AVAILABLE. Statusänderungen treten in der Regel dann auf, wenn sich die Verfügbarkeit des Providers, während er aktiv ist, ändert (Signalabbruch o.Ä.).

```
@Override
public void onStatusChanged(String provider, int status, Bundle extras) {
}

   ////////////////////////////////////////////////////////////////////////
///////////////////
private static final int TWO_MINUTES = 1000 * 60 * 2;
```

Mit dieser Methode werden zwei Locations miteinander verglichen und, nach gewissen Kriterien ausgewählt, die bessere Location genommen. Es ist nämlich nicht garantiert, dass die aktuell gelieferte Location genauer oder besser ist als die zuletzt übermittelte Location.

Listing 4.50: **LocationManagement, eine wiederverwendbare Klasse zur Positionsbestimmung**

```
protected boolean isBetterLocation(Location location, Location currentBest➥
    Location) {
    if (currentBestLocation == null) {
    // A new location is always better than no location
        return true;
    }

    // Check whether the new location fix is newer or older
    long timeDelta = location.getTime() - currentBestLocation.getTime();
    boolean isSignificantlyNewer = timeDelta > TWO_MINUTES;
    boolean isSignificantlyOlder = timeDelta < -TWO_MINUTES;
    boolean isNewer = timeDelta > 0;

    // If it's been more than two minutes since the current location, use
the new location
    // because the user has likely moved
    if (isSignificantlyNewer) {
        return true;
        // If the new location is more than two minutes older, it must be
worse
    } else if (isSignificantlyOlder) {
        return false;
    }

    // Check whether the new location fix is more or less accurate
    int accuracyDelta = (int) (location.getAccuracy() - currentBestLocation.
getAccuracy());
    boolean isLessAccurate = accuracyDelta > 0;
    boolean isMoreAccurate = accuracyDelta < 0;
    boolean isSignificantlyLessAccurate = accuracyDelta > 200;

    // Check if the old and new location are from the same provider
    boolean isFromSameProvider = isSameProvider(location.getProvider(),
    currentBestLocation.getProvider());

    // Determine location quality using a combination of timeliness and ac➥
    curacy
    if (isMoreAccurate) {
        return true;
    } else if (isNewer && !isLessAccurate) {
        return true;
    } else if (isNewer && !isSignificantlyLessAccurate && isFromSameProvi➥
    der) {
        return true;
    }
    return false;
}

/** Checks whether two providers are the same */
private boolean isSameProvider(String provider1, String provider2) {
if (provider1 == null) {
return provider2 == null;
}
        return provider1.equals(provider2);
}


}
```

Im ScrapBook wird diese Klasse nun benutzt, um an ein Scribble ein Geo-Tag anzuhängen:

Listing 4.51: **Benutzen der Klasse im ScrapBook**

```
[…]
public void startGeoTagging()
{
    locationManagement.start(true,this);
}

public void stopGeoTagging()
{
    locationManagement.stop();
}

@Override
public void betterFixAvailable(Location location) {
if (location!=null)
{
    currentLatitude = location.getLatitude();
    currentLongitude = location.getLongitude();
    hasLocation = true;
    postInvalidate();
    updateStatus();
}

}
```

Durch den `betterFixAvailable(Location location)`-Callback erhalten wir aus der Klasse den letzten, für gut befundenen, Fix und speichern die geografische Länge und geografische Breite in unserem ScribbleWidget. Beim Speichern des Scribbles wird dieses Geo-Tag dann in die Datenbank geschrieben. Hierfür haben wir zwei Float-Felder in der Tabelle angelegt, die wir beim Speichern besetzen:

Listing 4.52: **Abspeichern der geografischen Länge und Breite**

```
values.put(Scrap.Columns.SCRAP_LONGITUDE,this.currentLongitude);
values.put(Scrap.Columns.SCRAP_LATITUDE,this.currentLatitude);
```

INFO

Da die vorherige Version 1 der Datenbank diese Felder noch nicht hatte, habe ich im Content Provider die Methode `public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)` **überschrieben, respektive in meinem prototypbasierten Ansatz die Methode** `upgradeTable(…)`**:**

```
public void upgradeTable(SQLiteDatabase db, int oldVersion, int newVersion)
{
    db.execSQL("ALTER TABLE " + Scrap.TABLE_NAME + " add "+Scrap.Columns.
SCRAP_LATITUDE+" FLOAT");
    db.execSQL("ALTER TABLE " + Scrap.TABLE_NAME + " add "+Scrap.Columns.
SCRAP_LONGITUDE+" FLOAT");
}
```

Damit können die Scribbles nun mit dem Ort verbunden werden, an dem sie entstanden. Um den Ort nun in Google Maps anzuzeigen, bedienen wir uns eines speziellen Intents:

Listing 4.53: **Anzeigen der Position in Google Maps**

```
protected void showInMaps()
{
    if (getScribbleWidget().hasLocation())
    {
        Intent intent = new Intent(Intent.ACTION_VIEW);
        String s = "geo:"+getScribbleWidget().getCurrentLatitude()+","+get-
ScribbleWidget().getCurrentLongitude()+"?z=20";
        intent.setData(Uri.parse(s));
        startActivity(intent);
    }
    else
    {
        Toast.makeText(this, "Das Scribble hat kein Geotag", Toast.LENGTH_
SHORT).show();
    }
}
```

Das Entscheidende ist der Aufbau des URI. Die Form lautet `geo:<latitude>,<longitude>?z=<zoom>`. Das `geo`-Schema befindet sich unter dem Titel *A Uniform Resource Identifier for Geographic Locations* bei der IETF (Internet Engineering Task Force) im Draft-Status und wird von Google Maps auch noch nicht vollumfänglich unterstützt. Wer Interesse hat, das Draft ist unter `http://tools.ietf.org/html/draft-mayrhofer-geo-uri-00` zu finden.

Wir bauen also den URI aus der geografischen Länge und Breite auf und hängen noch einen Parameter z = 20 an, was in Google Maps ein angenehmer Ausschnitt um die Zielkoordinate herum ergibt.

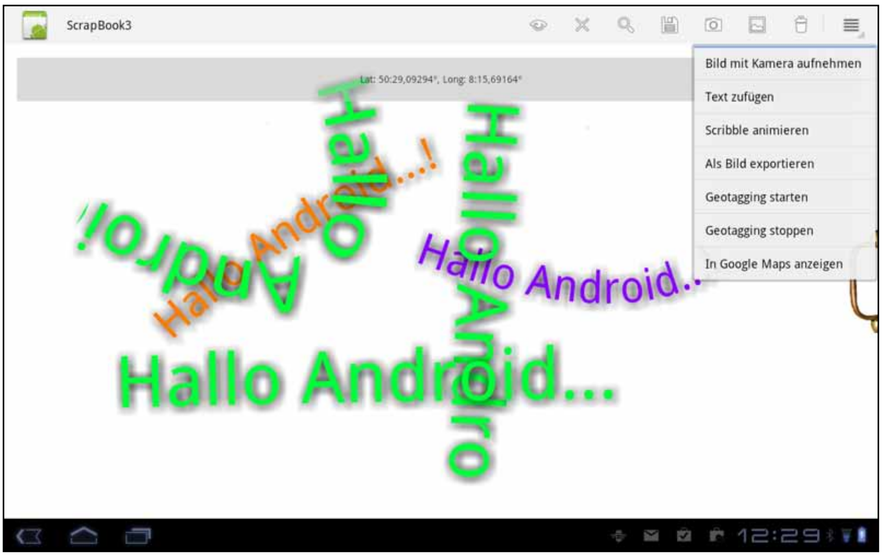


Abbildung 4.11: **Geo-Tagging eines Scribbles**

Die geografische Länge und Breite zeigen wir hier in einem Overlay an. Im Menü haben wir die Möglichkeit, den Ort in Google Maps zu suchen:

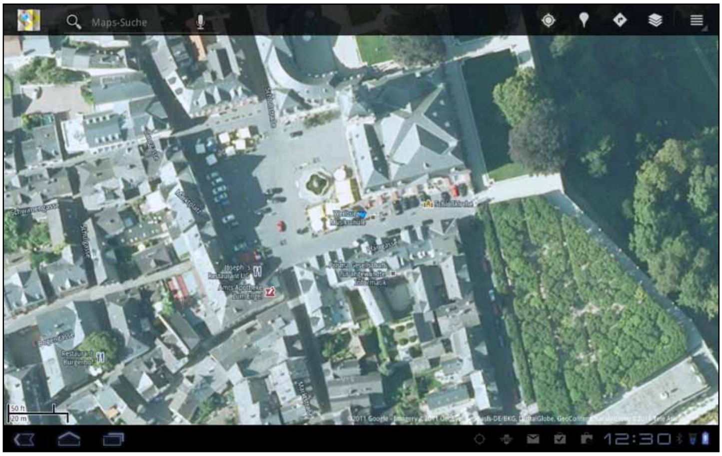


Abbildung 4.12: **Anzeige der Position in Google Maps**

Der LocationManager bietet noch einige interessante Dinge, neben dem Aufzählen der Satelliten über den GpsStatus.Listener bis hin zum Setzen eines Alarms per `locationManager.addProximityAlert(double latitude, double longitude, float radius, long expiration, PendingIntent intent)`, der das entsprechende Intent auslöst, wenn wir uns in den Dunstkreis einer Koordinate begeben.

INFO

Man muss nicht selbst auf die Locations hören, um diese Methode zu benutzen. Wir können z.B. die Koordinaten aus einer Datenbank hinzufügen und bestimmte Aktionen ausführen lassen, wenn man in die Nähe einer dieser Koordinaten kommt.

4.6 Multimedia

Android-Geräte sind wahre multimediale Talente. Leistungsfähige Prozessoren, in neueren Geräten werkeln Dual-Core-Prozessoren, hardwarebeschleunigte Grafik, OpenGL-Unterstützung, Kamerahardware und Audio-/Video-Komponenten machen die kleinen und mittelgroßen Geräte zu einer attraktiven Plattform für Spiele und multimediale Erlebnisse.

Die Kernkomponenten im Multimedia-Framework sind:

1. Die Klasse MediaPlayer
2. Die Klasse MediaRecorder
3. Die Klasse AudioManager
4. Die Klasse Camera
5. Die Klasse JetPlayer
6. Die Klasse RingtoneManager

Die Kamera haben wir bereits im Rahmen der SurfaceView kennengelernt, hier haben wir die Kameravorschau als Live-Hintergrund in das ScrapBook eingeblendet.

Das Abspielen von Multimedia-Inhalten ist mit dem MediaPlayer denkbar einfach. Per `MediaPlayer player = MediaPlayer.create(Context context, Uri uri)` lässt sich ein `MediaPlayer`-Objekt erstellen, das die durch den URI bezeichnete Datei abspielen kann. Das kann entweder ein Audi-File oder eine Video-Datei sein, wobei wir im Falle eines Videos noch einen SurfaceHolder übergeben müssen, auf dem das Video-Playback stattfinden kann: `MediaPlayer player = MediaPlayer.create(Context context, Uri uri, SurfaceHolder holder)`.

Das Einrichten des SurfaceHolder geschieht genau so, wie wir das für die Kamera-Vorschau durchgeführt haben.

Wichtig ist, dass wir den MediaPlayer wieder freigeben, wenn wir ihn nicht mehr benötigen:

```
player.release();
player = null;
```

Der MediaPlayer kann durch einige Methoden kontrolliert werden:

start()	Startet die Wiedergabe oder fährt mit der Wiedergabe fort
stop()	Stoppt die Wiedergabe
pause()	Pausiert die Wiedergabe
seekTo(int msec)	Springt an die entsprechende Stelle im Stream
setLooping(boolean loop)	Schaltet auf Wiederholung
setVolume(float left, float right)	Setzt die Lautstärke des linken und rechten Kanals

Tabelle 4.13: **Grundlegende Kontrollfunktionen**

ACHTUNG

Wenn es sich um große Dateien handelt, dann kann das Erstellen des Players mittels `create(…)` ggf. lange dauern, da automatisch `prepare()` aufgerufen wird. Entweder müssen wir den Player dann in einem eigenen Thread erstellen oder den Player selbst konstruieren und mit `prepareAsync()` die Initialisierung im Hintergrund laufen lassen.

Listing 4.54: **Asynchrones Initialisieren des Players**

```
final MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(this, musicFileUri);
    mediaPlayer.setOnPreparedListenet( new MediaPlayer.OnPreparedListener()
    {
        void onPrepared(MediaPlayer player)
        {
```

```
            player.start();
            }
        });
mediaPlayer.prepareAsync();
```

Diese Variante ist dem Schreiben eines eigenen Threads vorzuziehen, da sie weniger Aufwand bedeutet.

Wenn wir einen eigenen MediaPlayer bauen wollen, der die Musik auch dann abspielt, wenn die Activity schlafen geht, müssen wir den Player in einen Service verfrachten.

INFO

Wenn der MediaPlayer auf einen Fehler läuft, dann muss er mit `reset()` zurückgesetzt werden, bevor er erneut benutzt werden kann.

Listing 4.55: **MediaPlayer in einem Service**

```
public class MediaPlayerService extends Service implements MediaPlayer.On➥
    PreparedListener {
private static final ACTION_PLAY = "de.androidpraxis.action.PLAY";
private static final EXTRA_URI = " de.androidpraxis.EXTRA_URI";
MediaPlayer mediaPlayer = null;

public int onStartCommand(Intent intent, int flags, int startId) {
    if (intent.getAction().equals(ACTION_PLAY)) {
        mediaPlayer = new MediaPlayer();
        Uri musicFileUri = Uri.parse(intent.getStringExtra(EXTRA_URI));
        mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
        mediaPlayer.setDataSource(this, musicFileUri);
        mMediaPlayer.setOnPreparedListener(this);
        mMediaPlayer.prepareAsync();
    }
}
    public void onPrepared(MediaPlayer player) {
        player.start();
    }
}
```

Für das ScrapBook wiederum ist das Aufnehmen von Videos und Audio interessant. Zu diesem Zweck bietet das Framework den MediaRecorder, der mit der Kamera verknüpft werden oder auch nur Audio aufnehmen kann.

Um Audio aufzunehmen, müssen wir die Erlaubnis `<uses-permission android:name="android.permission.RECORD_AUDIO" />` im Manifest deklarieren, für Video entsprechend `<uses-permission android:name="android.permission.CAMERA" />`.

Wir wollen uns hier auf das Aufnehmen von Audiodaten beschränken, die an ein Scribble angehängt werden können.

Listing 4.56: **Aufnehmen einer Audiodatei**

```
public void startRecording() {
    recorder = new MediaRecorder();
    recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    String fileName = getContext().getExternalFilesDir("audio")+"/
audiosnippet_"+new Long(ContentUris.parseId(contentUri)).toString()+".3gp";
    recorder.setOutputFile(fileName);
    recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);

    try {
        recorder.prepare();
    } catch (IOException e) {
    Log.e(Globals.LOG_TAG, "prepare() failed");
    }

    recorder.start();
}

public void stopRecording() {
recorder.stop();
recorder.release();
recorder = null;
}
```

Das Abspielen können wir dann so realisieren:

Listing 4.57: **Abspielen des Audio-Snippets**

```
public void playAudioSnippet()
{
    String fileName = snippetFileName = getContext().
getExternalFilesDir("Audio/scrapbook")+"/audiosnippet_"+new
Long(ContentUris.parseId(contentUri)).toString()+".3gp";
    File f = new File(fileName);
    if (f.exists())
    {
    MediaPlayer player = MediaPlayer.create(getContext(), Uri.fromFile(f));
    player.setOnCompletionListener(new MediaPlayer.OnCompletionListener() {

        @Override
        public void onCompletion(MediaPlayer mp) {
        mp.reset();
        mp.release();
        }
        });
    player.start();

    }
}
```

Eine nützliche Sache ist der RingtoneManager, um Signaltöne abzuspielen. Bei der Near-Field-Communication-Anwendung in der Spielwiese benutze ich den RingtoneManager, um das Erkennen eines RFID-Tags zu signalisieren:

Listing 4.58: **Benutzen des RingtoneManagers**

```
Ringtone ringtone = RingtoneManager.getRingtone(this, RingtoneManager.getAct
ualDefaultRingtoneUri(this, RingtoneManager.TYPE_NOTIFICATION));
if (ringtone!=null) ringtone.play();
```

Der JetPlayer dient dazu, Musik in verschiedenen Tracks und als einzelne Bestandteile zu verwalten, die unabhängig voneinander getriggert werden können. Das ist in der Spieleentwicklung wichtig, wo zum einen Loops eingesetzt werden, die die Hintergrundmusik bilden, aber auch einzelne Tonsequenzen abhängig von der Objektbewegung, Kollisionen oder anderen Ereignissen benötigt. Das JET-Format und die Erstellung von entsprechenden Tracks ist allerdings nochmals ein Kapitel für sich und geht über den Rahmen hier hinaus.

Die Kamera ist natürlich ein spannendes Gerät. Wenn wir die Kamera benutzen, und wir haben einige Möglichkeiten bereits in der Spielwiese und im ScrapBook gesehen, müssen wir uns entscheiden, was wir benötigen:

1. Aufnehmen und weiterverarbeiten von Bildern über die eingebaute Kameraanwendung
2. oder eine eigene Kamerasteuerung, um spezielle Anforderungen zu implementieren oder die eingebaute Kameraapplikation zu »ersetzen«.

Wenn wir eine Bildbearbeitung vornehmen oder auch eine Anwendung wie das ScrapBook bauen, dann ist es im Grunde nicht nötig, die Kamera direkt anzusteuern. Dann können wir einfach die eingebaute Kameraanwendung benutzen und das aufgenommene Bild in unserer Anwendung verwenden.

Zu diesem Zweck nutzen wir ein entsprechendes Intent:

Listing 4.59: **Erstellen eines URI für eine Datei, in die die Kamera die Aufnahme ablegen kann**

```
public Uri createDestinationUri(String bucket, String filename)
{
    if (!this.isExternalStorageAvailable())
    {
        return null;
    }
    if (!this.isExternalStorageWriteable())
    {
        return null;
    }

    Uri result = null;

    String path = Environment.DIRECTORY_PICTURES;
    if (bucket!=null && !bucket.equals(""))
    {
        path = path +"/" + bucket;
    }
    File picturePath = context.getExternalFilesDir(path);
    File file = new File(picturePath, filename );
    result = Uri.fromFile(file);
    return result;
}
```

Listing 4.60: **Fotografieren eines Bildes**

```
[…]
public final static int BACKGROUND_IMAGE_CAPTURE = 1;
[…]
private void captureImageFromCamera() {

    StorageHelper helper = new StorageHelper(this);
    DateFormat df = new DateFormat();
    String isodate = df.format("yyyyMMdd-hhmmss", Calendar.getInstance().
getTime()).toString();

    captureUri = helper.createDestinationUri("ScrapBook", isodate+".jpg");
    if (captureUri!=null)
    {
        Intent picImage = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        picImage.putExtra(MediaStore.EXTRA_OUTPUT, captureUri);

        startActivityForResult(picImage,BACKGROUND_IMAGE_CAPTURE);
    }
}
```

ACHTUNG

Der URI der Datei, unter der die Kamera das Bild speichern soll, muss eine Datei adressieren, die öffentlich zugänglich ist, also entweder mit `MODE_WORLD_WRITEABLE` erstellt wurde oder in `context.getExternalFilesDir(…)` abgelegt werden soll. Ansonsten kann die Kamera das Bild nicht anlegen.

Um das fotografierte Bild nun zu benutzen, müssen wir die Rückkehr der Activity abwarten und das Ergebnis verwerten:

```
protected void onActivityResult(final int requestCode, final int resultCode,
final Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    if (data!=null)
    {
```

Der Parameter `data` ist belegt, wenn wir z.B. ein Bild aus der Galerie auswählen.

```
        if (requestCode==BACKGROUND_IMAGE_CAPTURE)
        {
        getScribbleWidget().setBackgroundImageContentUri(data.getData());
        }
    }
    else
    {
```

Der Parameter `data` ist **nicht** belegt, wenn wir per Kamera fotografieren.

Listing 4.61: **Übernehmen des Bildes**

```
    if (requestCode==BACKGROUND_IMAGE_CAPTURE)
    {
```

```
        if (captureUri!=null)
        {
        getScribbleWidget().setBackgroundImageContentUri(captureUri);
        captureUri = null;
        }
    }
    }
    if (resultCode == Activity.RESULT_CANCELED)
    {

    }
}
```

ACHTUNG

Wir müssen uns den URI des Ziels innerhalb unserer Activity merken, da der URI selbst bei der Rückkehr der Activity nicht mehr übergeben wird.

Wir müssen ebenso einen URI angeben! Wenn wir das nicht machen, dann liefert die Kamera nur ein Thumbnail der Aufnahme zurück. Das könnte gewünscht sein, wenn wir in unserer Anwendung sowieso nur die Vorschau benötigen – im ScrapBook will ich aber das komplette Bild.

Wenn wir dennoch nur das Thumbnail benötigen, dann können wir das folgendermaßen realisieren:

Listing 4.62: **Extrahieren des Thumbnails**

```
[…]
Intent picImage = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
startActivityForResult(picImage,BACKGROUND_IMAGE_CAPTURE);
[…]
@Override
protected void onActivityResult(final int requestCode, final int resultCode,
final Intent data) {
super.onActivityResult(requestCode, resultCode, data);

    if (data!=null)
    {
        Bitmap bmp = null;
        if (data.hasExtra("data")) bmp = data.getParcelableExtra("data");
    }
    […]
}
```

Mit dieser Methode brauchen wir uns keine weiteren Gedanken um die Kamerasteuerung zu machen. All das wird von der Kameraanwendung (fast) perfekt abgebildet. Allerdings können wir dann auch einige spannende Spielereien nicht machen.

Vielleicht wollen wir ja im Vorschaubild nach Gesichtern oder anderen interessanten Objekten suchen oder, wie im ScrapBook, live auf der Vorschau malen, wo, um das Ganze perfekt zu machen, auch die Bewegung erkannt und das Gemalte nachgeführt werden könnte.

Egal wie komplex die weiteren Anforderungen sind, die eigene Kamerasteuerung setzt bei der Klasse `Camera` in Verbindung mit einer SurfaceView an, auf der wir die Kameravorschau abspielen können.

INFO

Die natürliche Ausrichtung der Kamera ist immer die Längsseite des Geräts. Das heißt, bei einem Hochkantgerät ist der »natürliche« Betriebsmodus im Querformat, das Gerät müsste mithin gedreht werden, ansonsten fotografiert man halt hochkant. Bei querformatigen Geräten ist die natürliche Ausrichtung gleich der natürlichen Kameraausrichtung.

Welche Konsequenzen hat das? Wir müssen uns überlegen, in welcher Ausrichtung unsere Anwendung betrieben wird. Ist die Betriebsart im Hochkantformat und das Gerät ist ein hochkantformatiges Gerät, dann müssen wir die Kamera um 90° drehen. Ist das Gerät im Querformat und die Betriebsart der Anwendung ebenso, müssen wir die Kamera nicht drehen.

Dreht sich unsere Anwendung, müssen wir die Kamera entsprechend korrigieren.

Wenn wir die Kamerasteuerung selbst übernehmen wollen, dann müssen wir, je nachdem, was wir alles machen wollen, auch entsprechende Rechte im Manifest deklarieren:

Listing 4.63: **Rechte zum Zugriff auf die Kamerahardware**

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

TIPP

Die Erlaubnis für RECORD_AUDIO benötigen wir nur dann, wenn wir Videos mit Ton aufzeichnen wollen. Für eine reine Foto-Anwendung benötigen wir das nicht.

Das Wichtigste beim Schreiben einer Kameraanwendung ist zum einen die SurfaceView, auf der die Vorschau stattfinden soll, und zum anderen das Auslesen und Setzen der gewünschten Parameter. Im folgenden Listing sehen wir eine Grundlage für eine Kameravorschau, die in der Spielwiese und im ScrapBook verwendet wird. Wichtige Elemente werden im Folgenden erörtert.

```
public class CameraView extends SurfaceView implements
    SurfaceHolder.Callback,
    Camera.PreviewCallback,
    Camera.ErrorCallback,
    Camera.AutoFocusCallback,
    Camera.OnZoomChangeListener,
    Camera.PictureCallback,Camera.ShutterCallback {
[…]
private void initView()
{
    camera = null;
    surfaceHolder = getHolder();
    surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS); //Laut
Doku "deprecated", muss aber in diesem Kontext gesetzt werden, sonst gibts
vor 3.0 Probleme mit dem Preview (Absturz)!!
```

Damit kümmert sich die View selbst um Änderungen an der darunter liegenden Surface. Auf eine Surface darf nur zugegriffen werden, wenn sie korrekt initialisiert ist. Das wissen wir durch die Callbacks, die z.B. aufgerufen werden, wenn die Surface erstellt wurde.

```
    surfaceHolder.addCallback(this);
}
protected Camera openCameraImplementation()
{
```

Hier öffnen wir die Kamera und liefern sie zurück.

```
    return Camera.open();
}
```

Diese Methode prüft erst einmal, ob schon eine Kamera geöffnet wurde. Wenn nicht, dann wird die Kamera geöffnet und bei Bedarf noch an die Orientierung der Anwendung angepasst.

```
private void openCamera()
{
    if (camera==null)
    {
    camera = openCameraImplementation();
    cameraParameters = camera.getParameters();
    int rotation = ((WindowManager)getContext().getSystemService(Context.
WINDOW_SERVICE)).getDefaultDisplay().getRotation();
    int degrees = 0;
    int w = ((WindowManager)getContext().getSystemService(Context.WINDOW_
SERVICE)).getDefaultDisplay().getWidth();
    int h = ((WindowManager)getContext().getSystemService(Context.WINDOW_
SERVICE)).getDefaultDisplay().getHeight();
```

Hier kommt die Prüfung auf die Ausrichtung der Anwendung gegenüber der natürlichen Ausrichtung, die Drehung der Kamera wird entsprechend gesetzt.

```
    if (DisplayManagement.naturalOrientation(getContext()) == Configuration.
ORIENTATION_LANDSCAPE)
    {
        switch (rotation) {
        case Surface.ROTATION_0: degrees = 0; camdegrees = 0; break;
        case Surface.ROTATION_90: degrees = 90; camdegrees = 270; break;
        case Surface.ROTATION_180: degrees = 180; camdegrees = 180; break;
        case Surface.ROTATION_270: degrees = 270; camdegrees = 90; break;
        }
    }
    else
    {
        switch (rotation) {
        case Surface.ROTATION_0: degrees = 0; camdegrees = 90; break;
        case Surface.ROTATION_90: degrees = 90; camdegrees = 0; break;
        case Surface.ROTATION_180: degrees = 180; camdegrees = 270; break;
```

```
        case Surface.ROTATION_270: degrees = 270; camdegrees = 180; break;
    }
    }
```

Hier wird die Kamera gedreht.

```
    if (rotate) camera.setDisplayOrientation(camdegrees);

    int orientation = getResources().getConfiguration().orientation;
    Log.d(Globals.LOG_TAG, "CameraView.openCamera() orinetation "+orientati➥
    on+" rotation "+degrees+" "+camdegrees);

    DisplayMetrics outMetrics = new DisplayMetrics();
    ((WindowManager)getContext().getSystemService(Context.WINDOW_SERVICE)).
getDefaultDisplay().getMetrics(outMetrics);
    Log.d(Globals.LOG_TAG, "CameraView.openCamera() outMetrics "+outMetrics.
heightPixels+" "+outMetrics.widthPixels+" "+degrees);
```

Falls die Surface schon initialisiert wurde, setzen wir den SurfaceHolder als Vorschauanzeige der Kamera.

```
    if (state==STATE_INITIALIZED)
    {
    try {
        camera.setPreviewDisplay(surfaceHolder);
        camera.setPreviewCallback(this);
    } catch (IOException e) {
    }
    }

    }
}
```

Die Methode `setPreviewParameter` dient dazu, bestimmte Vorgaben zu setzen. Hier suchen wir die Vorschaugröße heraus, die am besten zur Abmessung des Widgets passt, und setzen den Autofocus-Modus. Mittels der Klasse `Camera.Parameters` und `camera.setParameters(parameters)` sind mannigfaltige Einstellungen möglich.

```
private void setPreviewParameter(Camera camera, Camera.Parameters params)
{
    List<Camera.Size> previewSizes = params.getSupportedPreviewSizes();
    long mydiag2 = getHeight()*getHeight() + getWidth()*getWidth();
    Camera.Size psize = null;
```

Wir durchsuchen alle möglichen Vorschaugrößen, die die Kamera bietet, und suchen uns die Größe heraus, deren Diagonale die nächstkleinere zur Diagonalen unseres Widgets ist.

```
    for(Camera.Size size : previewSizes)
    {
        long diag2 = size.height*size.height + size.width*size.width;
```

```
        if (diag2>=mydiag2)
        {
        break;
        }
        psize = size;
        }
        if (psize!=null)
        {

        params.setPreviewSize(psize.width, psize.height);

    }
```

Und wir schauen noch, welchen Autofokus-Modus die Kamera unterstützt.

```
    List<String> focusModes = params.getSupportedFocusModes();
    if (focusModes.contains(Camera.Parameters.FOCUS_MODE_EDOF))
    {
        params.setFocusMode(Camera.Parameters.FOCUS_MODE_EDOF);
    }
    else if (focusModes.contains(Camera.Parameters.FOCUS_MODE_AUTO))
    {
        params.setFocusMode(Camera.Parameters.FOCUS_MODE_AUTO);
    }
```

Hier können noch weitere Parameter gesetzt werden, z.B. der Blitz (FLASH_MODE_AUTO, FLASH_MODE_RED_EYE ...) und vieles andere mehr, da lohnt sich auf jeden Fall ein Blick auf die Online-Dokumentation.

```
    camera.setParameters(params);
}
```

Die nächsten Methoden dienen dazu, auf die Änderung der Surface zu reagieren, wenn sie erstellt wurde, wenn sie erzeugt wurde und wenn sie sich verändert hat. Eine Veränderung der Surface findet z.B. statt, wenn sich die Abmessungen des Widgets ändern.

```
public void surfaceChanged(SurfaceHolder holder, int format, int width, int
height) {

    Log.d(Globals.LOG_TAG, "CameraView.surfaceChanged()");
    if (holder.getSurface()==null)
    {
    return;
    }

    if (camera!=null)
    {
    try
    {
        camera.stopPreview();
    } catch (Exception e) {
    Log.d(Globals.LOG_TAG, "camera.setPreviewDisplay(holder)",e);
```

```
    }
    try
    {
        camera.setPreviewDisplay(holder);
        camera.startPreview();
    } catch (Exception e) {
    Log.d(Globals.LOG_TAG, "camera.setPreviewDisplay(holder)",e);
    }
    }
}
```

Wenn die Surface erstellt wurde, dann öffnen wir die Kamera bei Bedarf und setzen den SurfaceHolder ein.

```
public void surfaceCreated(SurfaceHolder holder) {

try {

openCamera();
if (camera!=null)
{
        Log.d(Globals.LOG_TAG, "CameraView.surfaceCreated()");
        camera.setPreviewDisplay(holder);
        camera.setPreviewCallback(this);
        setState(STATE_INITIALIZED);
}
else
{
        setState(STATE_INITIALIZATIONFAILED);
}
} catch (IOException e) {
Log.d(Globals.LOG_TAG, "camera.setPreviewDisplay(holder)",e);
}
}
public void surfaceDestroyed(SurfaceHolder holder) {

    try {
    Log.d(Globals.LOG_TAG, "CameraView.surfaceDestroyed()");
    if (camera!=null)
    {
        camera.setPreviewCallback(null);
        camera.setPreviewDisplay(null);
        camera.release();
        camera = null;
        surfaceHolder = null;
    }
    } catch (IOException e) {
    Log.d(Globals.LOG_TAG, "camera.setPreviewDisplay(holder)",e);
    }
}
```

Die Methode, um die Vorschau zu starten. Es wird versucht, die Vorschau direkt zu starten. Falls die Surface noch nicht initialisiert ist, dann wird ein Thread gestartet, der eine gewisse Zeit wartet, um der Surface die Chance zu geben, erstellt zu werden.

```
public void startPreview()
{
    openCamera();
    if (state == STATE_INITIALIZED)
    {
    if (camera!=null)
    {
    setPreviewParameter(camera,cameraParameters);
    camera.startPreview();
    }
    }
    else
    {
```

Dieser Thread dient dazu, den Preview-Start sozusagen so lange zu verzögern, bis die Surface initialisiert wurde.

```
    Thread delayedStart = new Thread()
    {
        @Override
        public void run()
        {
        while (state==STATE_UNINITIALIZED)
        {
        try {
        sleep(1000);
        } catch (InterruptedException e) {
        break;
        }
        }

        if (state==STATE_INITIALIZED)
        {
        post(new Runnable()
        {

        public void run()
        {
        if (camera!=null)
        {
        setPreviewParameter(camera,cameraParameters);
        camera.startPreview();
        }
        }
        });
    }
    }
    };
    delayedStart.start();
    }
}

public void stopPreview()
{
    if (camera!=null)
    {
```

```
        camera.stopPreview();
    }
}
```

Das Freigeben der Kamera ist wichtig, damit andere Anwendungen darauf zugreifen können.

Listing 4.64: **Grundlegende View für Kameravorschau**

```
public void releaseCamera()
{
    if (camera!=null)
    {
        camera.setPreviewCallback(null);
        camera.release();
        camera = null;
    }
}
[…]
}
```

Um nun ein Bild aufzunehmen, müssen wir nur camera.takePicture(…) aufrufen und einen entsprechenden Callback übergeben. Je nachdem, welche Callbacks übergeben werden, können wir die Rohdaten oder aber auch die JPEG-Daten abfangen:

```
public class CameraView extends SurfaceView implements
    SurfaceHolder.Callback,
    Camera.PreviewCallback,
    Camera.ErrorCallback,
    Camera.AutoFocusCallback,
    Camera.OnZoomChangeListener
Camera.PictureCallback,Camera.ShutterCallback {
[…]
public void takePicture()
{
    if (getCamera()!=null)
    {
```

Der dritte Parameter definiert den JPEG-Callback.

```
        getCamera().takePicture(this, null, this);
    }
}
```

Und hier kommt der JPEG-Callback:

```
@Override
public void onPictureTaken(byte[] data, android.hardware.Camera camera) {
    DateFormat df = new DateFormat();
    String isodate = df.format("yyyyMMdd-hhmmss", Calendar.getInstance().
getTime()).toString();
```

Hier speichern wir das Bild ab. Der Storage-Helper legt es unterhalb des PICTURE-Verzeichnisses in einem neuen Album an und schickt den MediaScanner los, das Bild auch in die Galerie mit aufzunehmen.

```
    StorageHelper storageHelper = new StorageHelper(getContext());
    storageHelper.saveJPEGBuffer("Mein Album", isodate+".jpg", data,
    new MediaScannerConnection.OnScanCompletedListener() {
        public void onScanCompleted(String path, Uri uri) {
    }
    });
```

Nach Aufnahme des Bildes starten wir die Vorschau wieder, da die sie durch das Auslösen der Aufnahme gestoppt wird.

Listing 4.65: **Aufnehmen und Abspeichern eines Bildes**

```
    getCamera().startPreview();
}
[…]
}
```

INFO

Das Auslösen von `takePicture(…)` ist nur bei laufender Vorschau möglich.

Ein interessantes Callback für weitere Anwendungen ist das `onPrevieFrame(…)`-Callback, das die aktuellen Vorschaudaten übergeben bekommt. Hier könnten wir ansetzen, um z.B. irgendwelche Objekte in der Vorschau direkt zu identifizieren, Anwendungen wie Wordshot machen das z.B., um Text in der Vorschau zu finden und direkt zu übersetzen.

Ab Android 4 ist auch der Face-Detektor in das Kamera-Framework mit eingebaut, vorher gab es nur Klassen, um in Bitmaps Gesichter zu entdecken.

Aufbauend auf dem obigen Beispiel lassen sich sehr schön eigene Kameraanwendungen bauen. In der Spielwiese z.B. ist eine Kameravorschau enthalten, die in das Vorschaubild noch die Blickrichtung per Overlay-Widget einblendet.

4.7 Netzwerk

Über Wireles-LAN und Mobilfunk (GSM/EDGE/UMTS) lässt sich unser Gerät in das allgegenwärtige Internet bzw. per Wireless-LAN auch in ein Firmennetz oder ein privates Netz zu Hause einklinken. Wenn ein Gerät als Hotspot dienen kann, dann lassen sich auch Android-Geräte per W-LAN mit anderen Geräten in einem Netz zusammenschließen.

Das Android-Gerät erhält in jedem Fall eine (temporäre) IP-Adresse durch den HotSpot bzw. das PPP-Peer zugewiesen und kann damit vollständig über TCP/IP mit anderen Diensten kommunizieren.

Über diese TCP/IP-Verbindung laufen dann auch alle »Cloud«-Services, bei denen eine Synchronisation mit Diensten wie Facebook, Google Picasa oder auch Google Mail stattfindet.

TCP/IP-Kommunikation findet auf der untersten Ebene über Sockets statt, die eine Verbindung zwischen zwei Diensten darstellen und über die Daten übertragen werden. Das Android-Framework bietet hier aus dem Apache Harmony-Projekt die `javax.net`-Klassen, mit denen Anwendungen gebaut werden können, die über Sockets miteinander kommunizieren. Basierend auf den Sockets liefert das Framework noch weitere Klassen mit, die Protokolle auf einer höheren Ebene abbilden. So stellt das Framework einen Teil der Jakarta Commons-Bibliothek für die HTTP-Kommunikation (Hyper Text Transfer Protocol) in `org.apache.http` zur Verfügung, mit der Datentransfer über das HTTP-Protokoll abgewickelt werden kann.

Mit dem HTTP-Protokoll können Daten von und zu Webservern bzw. Webservices übertragen und somit eine große Bandbreite von Diensten angesprochen werden. Sowohl reine Webseiten als auch Webservices werden ja über HTTP-Server im Internet oder im Firmennetz veröffentlicht.

Um mit der Anwendung auf das Netz zuzugreifen, müssen wir die entsprechende Erlaubnis im Manifest reklamieren:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Im MarbleGame benutzen wir diese Form der Netzwerkkommunikation dazu, die Bitmaps für die SandwichLayer des Spielfelds in einem Loader-Thread zu laden:

Listing 4.66: **Abrufen einer Ressource per HTTP-GET-Request**

```
private HttpClient httpClient()
{
    if (httpClient==null)
    {
        httpClient = new DefaultHttpClient();
    }
    return httpClient;
}

public Bitmap loadBitmap(Uri uri)
{
    Bitmap result = null;

    HttpGet get = new HttpGet(uri.toString());
    try {
        HttpResponse response = httpClient().execute(get);
        if (response.getStatusLine().getStatusCode() == HttpStatus.SC_OK)
        {
        HttpEntity entity = response.getEntity();
        InputStream is = entity.getContent();
        result = BitmapFactory.decodeStream(is);
        }
        else
        {
        if (messageHandler!=null) messageHandler.onError(response.getStatus➥
```

```
    Line());
        }
    } catch (ClientProtocolException e) {

    if (messageHandler!=null)
    {
        messageHandler.onException(e);
    }
    } catch (IOException e) {
        if (messageHandler!=null) messageHandler.onException(e);
    }
    return result;
}
```

Über das Loader-Framework wird dieser Request im Hintergrund ausgeführt, sodass die Anwendung nicht blockiert, solange die Daten gelesen werden.

Diese Form des Requests kann man für viele Anforderungen gebrauchen, bei denen man Daten aus dem Netz saugen will. Wetterdaten erhält man z.B. bei einigen öffentlichen Webservices und kann per HTTP-Request die aktuellen Wetterdaten, manchmal auch die Wettervorhersage, als XML-Datenstrom oder RSS-Feed abholen und in seiner Applikation verwenden.

ACHTUNG

Bei der Nutzung von Webservices müssen unbedingt die Geschäftsbedingungen beachtet werden! Die Daten werden in der Regel nur für den persönlichen, nichtkommerziellen Gebrauch zur Verfügung gestellt.

Ein weiterer Aspekt ist der wechselseitige Datenaustausch, der z.B. für das Synchronisieren von Mail-Content und Kalenderdaten benötigt wird. Auch hierfür gibt es in der HTTP-Implementierung Methoden, um z.B. auch Daten an einen Webdienst mittels HTTP-PUT-Methode oder HTTP-POST-Methode zu übermitteln.

Viele aktuelle und, mit fortschreitender Verbreitung von Cloud-Services, zukünftige Dienste sind über HTTP ansprechbar und stellen ein auf XML oder dem JSON-Format basierendes Protokoll zum Datenaustausch bereit, sodass wir mit den http-Klassen über das Repertoire verfügen, mit dem wir die meisten Netzwerkaufgaben erledigen können.

Listing 4.67: **Übertragen einer Datei per HTTP-Post zu einem Webserver**

```
public void saveFileTo(Uri uri, String fileName, String contentType)
{
HttpPost post = new HttpPost(uri.toString());

try {
    File f = new File(fileName);
    FileEntity fileEntity = new FileEntity(f, contentType);
    post.setEntity(fileEntity);
    HttpResponse response = httpClient().execute(post);
    if (response.getStatusLine().getStatusCode() == HttpStatus.SC_OK)
    {
    }
    else
```

```
    {
    if (messageHandler!=null) messageHandler.onError(response.getStatus➥
    Line());
    }
} catch (ClientProtocolException e) {

if (messageHandler!=null)
{
messageHandler.onException(e);
}

} catch (IOException e) {

if (messageHandler!=null) messageHandler.onException(e);

}
}
```

Wenn wir andere Protokolle abbilden müssen, dann stehen uns die Sockets zur Verfügung bzw. Bibliotheken, die auf Socket-Basis bestimmte Protokolle abbilden.

Eine weitere Form des Netzwerks sind Piconetze über Bluetooth, mit denen Kleinstnetze zwischen Bluetooth-fähigen Geräten geknüpft werden können. Über Bluetooth lassen sich verschiedene Peripheriegeräte wie Headsets, Lautsprecher, Tastaturen etc. anschließen, aber auch die Verbindung zweier Android-Geräte kann über Bluetooth erfolgen.

Auf der untersten Ebene bietet Bluetooth auch eine Socket-Implementierung, sodass die Programmierung hier ähnlich ist wie bei der Netzwerkprogrammierung über TCP/IP.

Eine zukünftige Anwendung für das ScrapBook wird die Übertragung eines Fotos von einem Smartphone auf ein Tablet sein, auf dem das ScrapBook läuft. Da das Fotografieren mit einem Tablet reichlich … merkwürdig aussieht, soll mit dem handlichen Smartphone per Bluetooth direkt ein Bild auf das ScrapBook geschickt werden, um es dort dann zu »augmentieren«. Geplant ist hier dann auch die Verbindung zur Kompass-Overlay-View, dass wir direkt die Blickrichtung mit übertragen.

Da das den Rahmen an dieser Stelle sprengen würde (und ich auch langsam mal zum Schluss kommen muss), möchte ich euch einladen, unter `www.androidpraxis.de` darauf zu warten, bis ich diesen Part veröffentlichen kann, um dann darüber mit mir zu diskutieren.

Das betrifft auch das Thema USB und auch einige weitere Themen, bei denen wir noch sehr viel tiefer einsteigen müssten.

4.8 Near-Field-Communication

Near-Field-Communication wurde mit Version 2.3 eingeführt und soll in Zukunft dazu dienen, über Nahfunk, und nah meint hier wirklich ganz nah, Daten zwischen Geräten auszutauschen. Das können Kontaktdaten, Bilder, Notizen etc. sein, ein großes Anwendungsgebiet wird aber wahrscheinlich die elektronische Geldbörse werden.

Durch die Beschränkung der Entfernung für die Datenübertragung auf wenige Zentimeter und ein sicheres (?) Protokoll sollen hierüber Transaktionen mit der elektronischen Geldbörse abgewickelt werden. Das ist auch gar nicht so von der Hand zu weisen, denn das Gerät ist ja in der Regel einem Besitzer zugeordnet, und dieser muss das Gerät auch aktiv an ein Bezahlterminal halten und den Vorgang initiieren.

Da die traditionellen RFID-Chips ebenfalls NFC-Geräte sind, ist ein weiteres Anwendungsgebiet das Beschreiben und Auslesen eben jener RFID-Chips. Mit diesen Chips kann man dann zusätzliche Informationen an irgendwelchen Dingen anbringen, nicht umsonst spricht man ja auch von elektronischen Etiketten (englisch: Tag).

Ich habe mir ein paar RFID-Etiketten besorgt, und zwar die Modelle *Mifare Ultralight* und *Mifare 1K*, um das Auslesen und Beschreiben von RFID-Tags zu realisieren.

Zur Anwendung kann diese Technik im Bereich der Smartposter kommen, aber ich kann es mir auch gut im Umfeld von Ausstellungen vorstellen, bei denen die Ausstellungsstücke mit einem RFID-Chip markiert sind. Vielleicht auch eine weitere Form der Augmented Reality, indem Sehenswürdigkeiten markiert werden. Auf dem Tag könnte dann z.B. ein weiterführender Link codiert sein, der uns direkt auf eine Infoseite oder zu Wikipedia bringt.

Grundlage des NFC-Datenaustauschs sind die NDEF-Nachrichten (NFC Data Exchange), die jeweils aus NDEF-Sätzen bestehen, von denen jeder in einem seiner Bestimmung gemäßen Format vorliegen muss, abhängig von der Technologie und der Verwendung.

Wir müssen die Nutzung der NFC-Technologie in unserem Manifest mit `<uses-permission android:name="android.permission.NFC" />` vereinbaren, und um unsere App im Market nur den Geräten zu präsentieren, die NFC können, vereinbaren wir noch die Eigenschaft `<uses-feature android:name="android.hardware.nfc" android:required="true" />`.

In der Regel versetzen wir zum Auslesen von NFC-Tags das Gerät in den Zustand, der sich *Discover* nennt, und bei Erkennen eines Tags wird ein Intent ausgelöst, das wir auswerten können und das die Daten das Tags (sofern vorhanden) transportiert.

Listing 4.68: **Initialisieren des Intents zur Tag-Discovery**

```
@Override
public void onCreate(Bundle savedState) {
    super.onCreate(savedState);
    […]
    mAdapter = NfcAdapter.getDefaultAdapter(this);
    mPendingIntent = PendingIntent.getActivity(this, 0,
    new Intent(this, getClass()).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP),
0);
    IntentFilter ndef = new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
    mFilters = new IntentFilter[] { ndef, };
    mTechLists = new String[][] { new String[] { NfcF.class.getName() } };
[…]
}
```

Wenn die Activity, mit der wir uns um die Tags kümmern wollen, aufgesetzt wird, holen wir uns eine Referenz auf den NFC-Adapter und erstellen ein PendingIntent, das beim Erkennen von Tags benutzt werden soll und eine Referenz auf unsere Activity erhält. Damit nicht bei jedem Tag die Activity neu gestartet wird, erhält das Intent das Flag `Intent.FLAG_ACTIVITY_SINGLE_TOP`.

Listing 4.69: **Aktivieren des Tag-Discovery als Vordergrundanwendung**

```
@Override
public void onResume() {
    super.onResume();
    mAdapter.enableForegroundDispatch(this, mPendingIntent, mFilters, mTech➥
    Lists);
}
```

In `onResume()` aktivieren wir das `foreGroundDispatch(...)`, damit wird unsere Anwendung zum aktiven »Entdecker«. Wenn das NFC-Subsystem ein Tag erkennt, und dieses Tag passt zu der Liste der von uns zu erkennenden Tag-Technologien, dann wird das entsprechende Intent aus dem PendingIntent gefeuert, das hier wiederum auf unsere Activity zeigt.

Im `onNewIntent(…)`-Callback unserer Activity reagieren wir nun auf neue Tags:

```
@Override
public void onNewIntent(Intent intent) {
    Log.i(Globals.LOG_TAG, "Discovered tag with intent: " + intent);
    if (intent.getAction()==NfcAdapter.ACTION_TAG_DISCOVERED)
    {
```

Wenn wir ein Tag erkannt haben, dann klingeln wir erst einmal.

```
    Ringtone ringtone = RingtoneManager.getRingtone(this, RingtoneManager.➥
getActualDefaultRingtoneUri(this, RingtoneManager.TYPE_NOTIFICATION));
    if (ringtone!=null) ringtone.play();
```

Jetzt extrahieren wir das eigentliche Tag:

```
    Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    String[] techList = tag.getTechList();
```

Und schreiben die Technologien des Tags raus.

```
    for (int i=0;i<techList.length;i++)
    {
        Log.i(Globals.LOG_TAG, "Technology: " + techList[i]);
    }
```

Wenn wir uns im Schreibmodus befinden, dann schreiben wir unseren Text auf das Tag:

```
    if (doWriteTag)
    {
        mText.setText("Auf Tag schreiben....");
        NfcUtilities.writeTag(tag, textToWrite);
        doWriteTag = false;
        textToWrite = "";
        mText.setText("Ein Tag auslesen...");
    }
    else
    {
```

Ansonsten probieren wir, die Inhalte des Tags auszulesen. Innerhalb der Klasse `NfcUtilities`, die in den Beispielen enthalten ist, können verschiedene Formate wie Smartposter o.Ä. geparst werden. Dieser Vorgang liefert eine Liste von NfcTextRecords zurück, die ebenfalls im Beispielcode vorhanden sind und aus denen dann einfacher Text extrahiert werden kann.

Listing 4.70: **Reagieren auf erkannte Tags**

```
        NdefMessage[] msgs = NfcUtilities.getNdefMessages(intent);
        if (msgs!=null)
        {
        Vector<NfcTextRecord> textrecords = NfcUtilities.
getParsedRecords(msgs);
        String result = "";
        Iterator<NfcTextRecord> iter = textrecords.iterator();

        while (iter.hasNext())
        {
        NfcTextRecord r = iter.next();
        result+=r.getText()+"\r\n"+"\r\n";
        }

        for (int i=0;i<techList.length;i++)
        {
        result+=techList[i]+"\r\n";
        }
        mText.setText(result);
    }
    }
}
```

Wie bei allen Ressourcen, hören wir auch hier mit dem Horchen auf, wenn die Anwendung schlafen geht:

Listing 4.71: **Horchen auf Tags beenden**

```
@Override
public void onPause() {
    super.onPause();
    mAdapter.disableForegroundDispatch(this);
}
```

Auf einen Tag schreiben können wir, wenn der Tag formatierbar ist oder wenn er bereits formatiert und nicht schreibgeschützt ist.

INFO

Die Logik zum Schreiben ist etwas kniffeliger. Zuerst einmal müssen wir, wie beim Auslesen auch, den Tag erkennen bzw. »discovern«. Dann wissen wir, dass wir einen Tag in Reichweite haben, kennen die Technologie des Tags und ob er formatierbar oder bereits beschrieben ist. In dem Moment, wo wir den Tag erkannt haben, können wir dann auch den Schreibvorgang durchführen. Daher versetze ich die Anwendung mit einem Flag in den Schreibmodus, wenn der Anwender auf den Knopf *Auf Tag schreiben* drückt, und übertrage den Inhalt in dem Moment, in dem der Tag in Reichweite, also »discovered« ist. Wenn ich jetzt erst noch die Bestätigung zum Schreiben einholen würde, wäre die Gefahr groß, dass der Anwender den Tag durch die Bewegung des Smartphones verliert.

Listing 4.72: **Schreiben auf ein Tag**

```
public static void writeTag(Tag t, String text)
{
try {
    Ndef tag = Ndef.get(t);
    if (tag==null)
    {
        formatTag(t,text);
        return;
    }
        Locale locale = Locale.US;
        NfcTextRecord tr = new NfcTextRecord(locale.getLanguage(),text);
        NdefRecord record = tr.getNdefRecord();
        if (record!=null)
        {
        NdefRecord[] records = {record};
        NdefMessage message = new NdefMessage(records);
        tag.connect();
        tag.writeNdefMessage(message);
        tag.close();
    }
}
catch (Exception e){
//do error handling
Log.d(Globals.LOG_TAG, "writeTag", e);
}
}
```

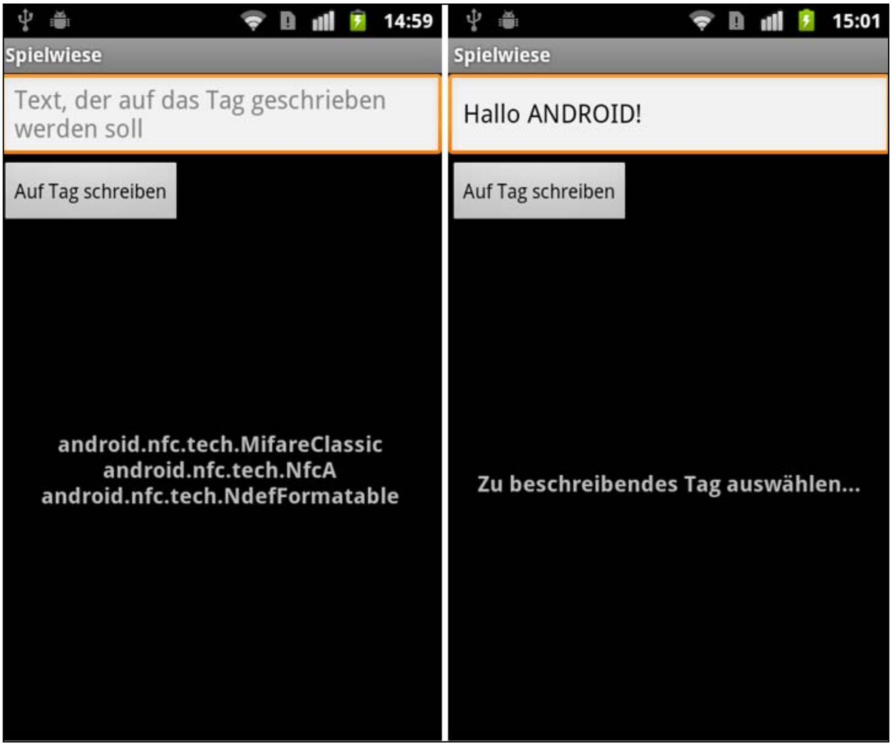


Abbildung 4.13: **Erkennen eines Tags und versetzen der Anwendung in den Schreibmodus**

Sobald der Text »Zu beschreibendes Tag auswählen ...« erscheint, muss der Anwender das zu beschreibende Tag ansteuern. Im Screenshot sehen wir links sehr schön, dass das Tag, das ich vorher angesteuert habe, formatierbar ist und somit noch keinen Inhalt aufweist.

Listing 4.73: **Formatieren eines Tags**

```
public static void formatTag(Tag t, String text)
{
try {
    NdefFormatable tag = NdefFormatable.get(t);
    if (tag==null)
    {
        Log.d(Globals.LOG_TAG, "writeTag: Tag nicht Beschreibbar");
        return;
    }
    Locale locale = Locale.US;
    NfcTextRecord tr = new NfcTextRecord(locale.getLanguage(),text);
    NdefRecord record = tr.getNdefRecord();
    if (record!=null)
    {
        NdefRecord[] records = {record};
        NdefMessage message = new NdefMessage(records);
        tag.connect();
        tag.format(message);
        tag.close();
```

```
    }
}
catch (Exception e){
//do error handling
Log.d(Globals.LOG_TAG, "writeTag", e);
}
}
```



Abbildung 4.14: **Schreiben auf das Tag und neuerliches Auslesen**

Das Tag ist beschrieben, und wir haben es neu ausgelesen.

Neben diesem passiven Modus kann ein Android-Gerät auch aktiv werden und Daten zu einem anderen übertragen (beamen :-)), wenn ein anderes entsprechendes Endgerät in der Nähe ist. Diese Technologie wird in Android 4 eingeführt und heißt »Android Beam™«.

Android 3 führt schon das *foreground NDEF pushing*, mit dem NDEF-Records auf andere NFC-Geräte geschoben werden können: `nfcAdapter.enableForegroundNdefPush (Activity activity, NdefMessage message)`. Damit wird die NdefMessage, die wir vorher erstellen müssen, dann an ein anderes NFC-Gerät »gepusht«, wenn unsere Activity im Vordergrund ist. Mit Android Beam™ wird sich in Version 4 hierzu noch einiges tun.

4.9 Veröffentlichen von Apps

Irgendwann ist es an der Zeit, die Anwendung(en), die man mit großem Enthusiasmus entwickelt hat, auch an die Frau und an den Mann zu bringen. War es in den Urzeiten des Heimcomputers noch gebräuchlich, Listings in Zeitschriften zum Abtippen zu veröffentlichen, auf Kassette zu spielen und irgendwann per Diskette weiterzugeben, sind die heutigen Distributionskanäle doch ungleich raffinierter und einfacher.

Android-Anwendungen werden über entsprechende elektronische Märkte verteilt, von denen die bekanntesten wohl der Google Android Market und der Amazon Appstore sind, wobei letzterer zurzeit nur den US-Kunden vorbehalten ist. Daneben gibt es noch weitere Market-Anbieter, wobei die größte Vielfalt im Google-Angebot zu finden ist.

INFO

Da Android ein, in weiten Teilen, offenes System ist, kann im Grunde jeder einen Market aufbauen. Das ist sicherlich ein großer Pluspunkt, da so eine relative Unabhängigkeit von einem Anbieter gewährleitet ist, der auch die Kontrolle besitzt. Andererseits wird manchmal die dadurch größere Gefahr, dass sich qualitativ schlechte oder gar schädliche Software einschleicht, kritisiert.

Um unsere Anwendung zu publizieren, muss sie beim Erstellen signiert werden. Zum Signieren benötigt man ein Zertifikat, das allerdings ein selbst ausgestelltes Zertifikat sein kann, eine (beglaubigte) Autorität ist nicht nötig.

Während des Entwickelns wird innerhalb der Eclipse die Anwendung immer automatisch mit dem Debug-Schlüssel signiert. Dieser ist für ein Veröffentlichen der Anwendung **nicht** geeignet.

TIPP

Alle Zertifikate haben ein Ablaufdatum, zu dem sie auslaufen. Das Debug-Zertifikat läuft nach 365 Tagen aus, und es kann dann vorkommen dass sich auch keine Debug-Versionen mehr installieren lassen. Wenn das passiert, kann man einfach den debug.keystore in seinem Benutzerverzeichnis, und hier im Unterverzeichnis `.android`, löschen. Dann wird ein neues Debug-Zertifikat erstellt.

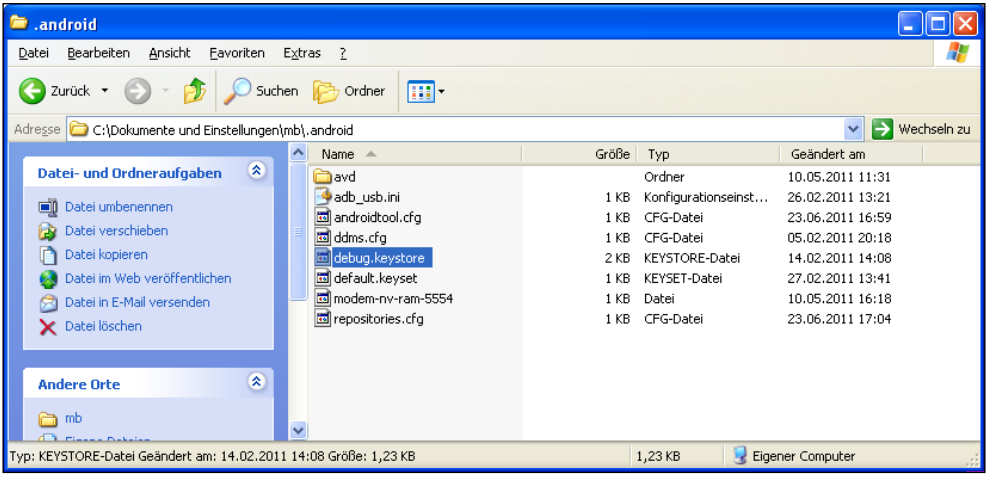


Abbildung 4.15: **Der Keystore in meinem Benutzerverzeichnis**

Für die Veröffentlichung wird eine Gültigkeitsdauer von 10.000 Tagen oder mehr empfohlen.

Bevor nun eine Anwendung veröffentlicht wird, muss/müssen

1. die Anwendung sorgfältig getestet sein, vor allem auf echten Geräten
2. die Anwendung korrekt versioniert werden
3. alle Debug-Ausgaben und das Logging ausgeschaltet sowie das android:debuggable-Attribut im Manifest auf »false« gesetzt werden
4. temporäre oder überflüssige Dateien gelöscht werden
5. die Anwendung korrekt signiert sein
6. die Anwendung per zipalign optimiert werden

Außerdem muss die Anwendung ein Icon und ein Label im Manifest deklarieren.

Die Versionierung findet durch die Angabe von `<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="de.androidpraxis.scrapbook3"` **android:versionName="2.0" android:versionCode="2"**`>` im Manifest statt. Es ist wichtig, bei einer neuen Version **beide** Angaben korrekt hochzusetzen, ansonsten können die Anwender kein Update der Anwendung fahren.

Label und Icon, die im Launcher und auch im Markt angezeigt werden, werden ebenfalls im Manifest deklariert: `<application` **android:icon="@drawable/icon" android:label="@string/app_name"** `android:theme="@android:style/Theme.Holo.Light">`. Es gibt einige Richtlinien, wie Icons gestaltet werden sollten, die wir auf der `http://developer.android.com/guide/practices/ui_guidelines/icon_design.html` finden.

In der Eclipse können wir unsere Anwendung sehr einfach als signiertes Package exportieren:

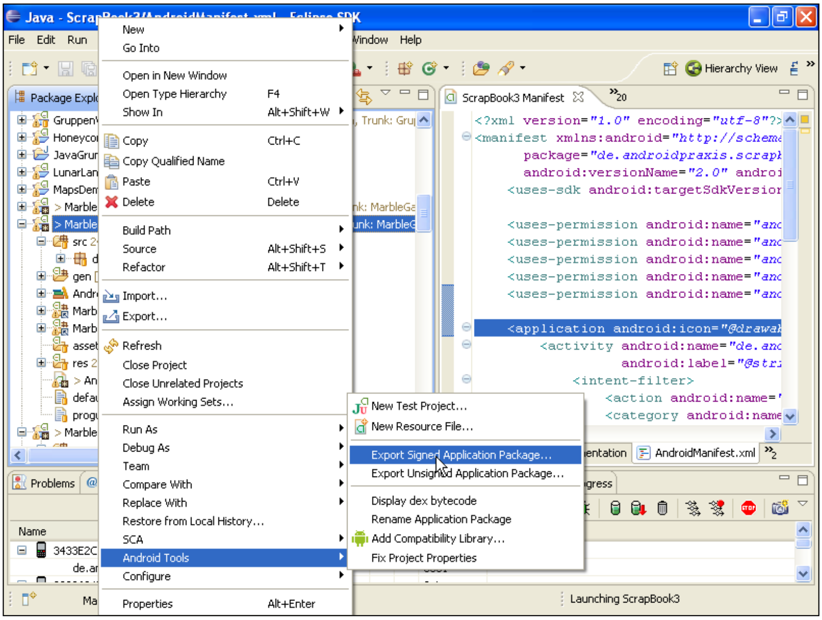


Abbildung 4.16: **Kontextmenü zum Exportieren des Anwendungspackages**

Nach dem Start des Exports werden wir durch die einzelnen Schritte geleitet:

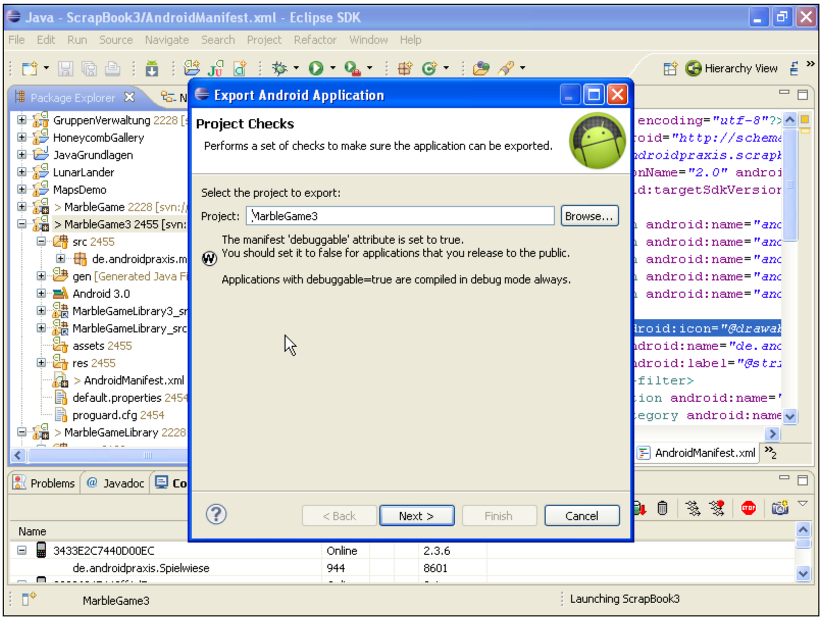


Abbildung 4.17: **Bestimmen, welches Projekt exportiert werden soll**

Hier ist schön zu sehen, dass ich vergessen habe, das Debuggable-Attribut auf false zu setzen. Das **müssen** wir vorher tun …

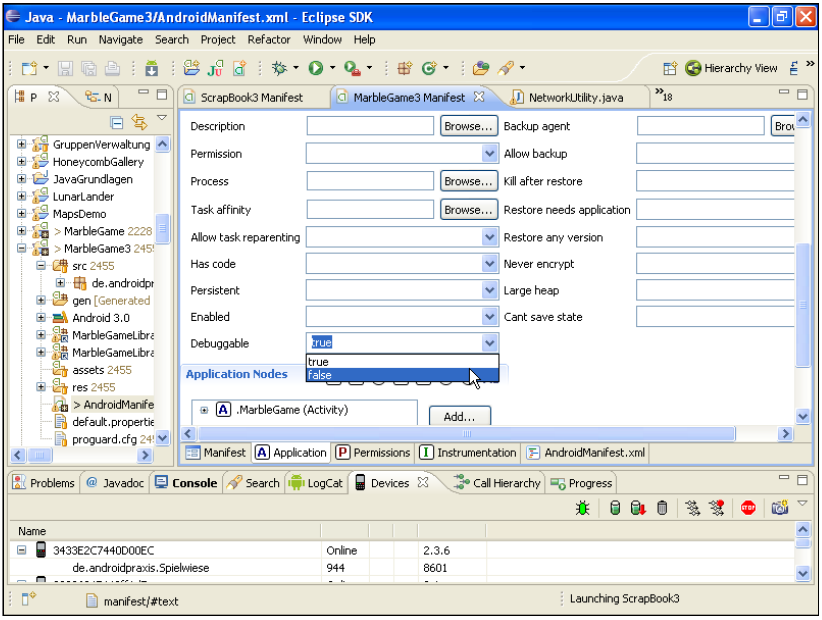


Abbildung 4.18: **Debuggable auf false setzen**

… und zwar hier. Jetzt geht es dann auch weiter:

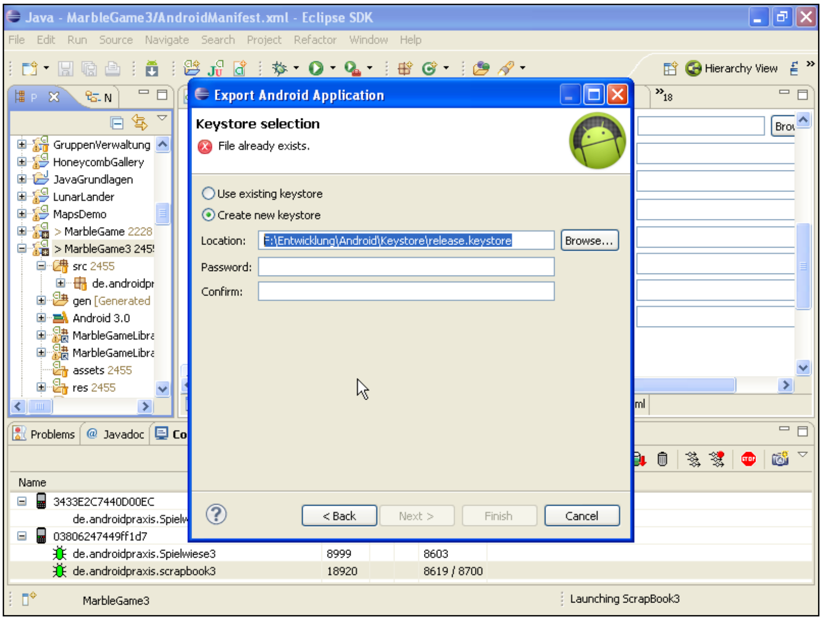


Abbildung 4.19: **Keystore auswählen**

Hier wählen wir den existierenden Keystore oder legen einen neuen an. Wenn wir einen neuen anlegen, dann müssen wir ein Passwort vergeben. Dieses Passwort müssen wir sicher verwahren, um später weitere Anwendungen oder neue Versionen mit diesem Zertifikat zu signieren.

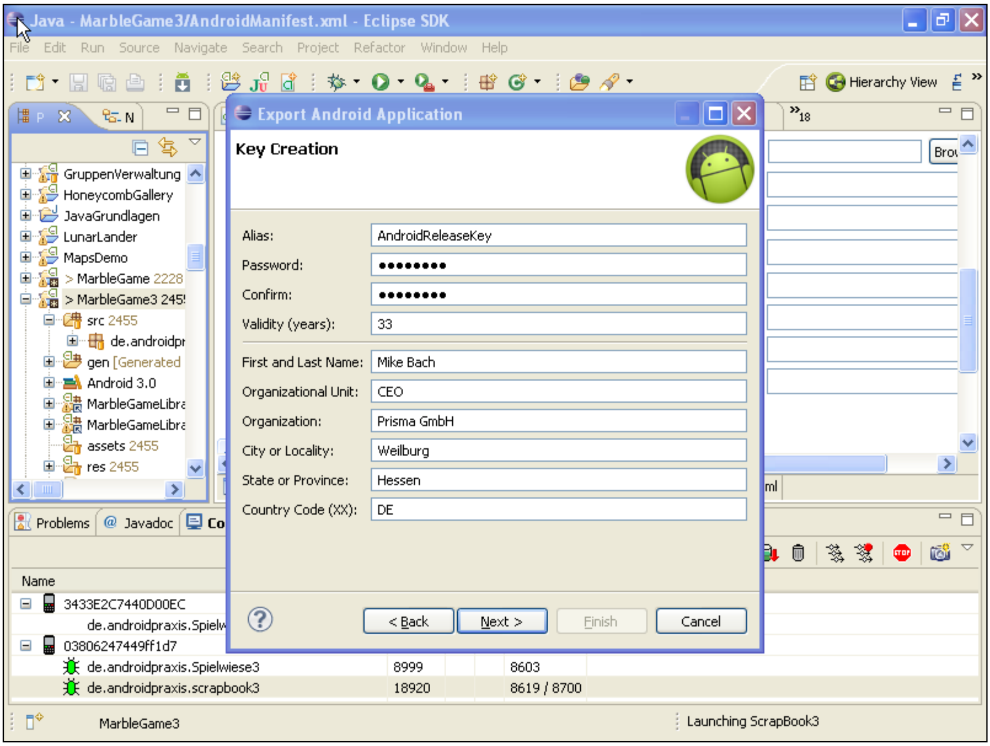


Abbildung 4.20: **Angabe der Informationen für das Zertifikat**

Der Alias ist der Name, mit dem wir später das Zertifikat zu weiteren Signierungsprozessen benennen.

Das Passwort hier dient zum Verschlüsseln des Zertifikats. In einem Keystore können mehrere Zertifikate abgelegt werden, die jeweils für sich unterschiedliche Schlüssel haben.

Die Gültigkeitsdauer habe ich hier mit 33 Jahren angegeben, das ist größer als der mindestens empfohlene Wert.

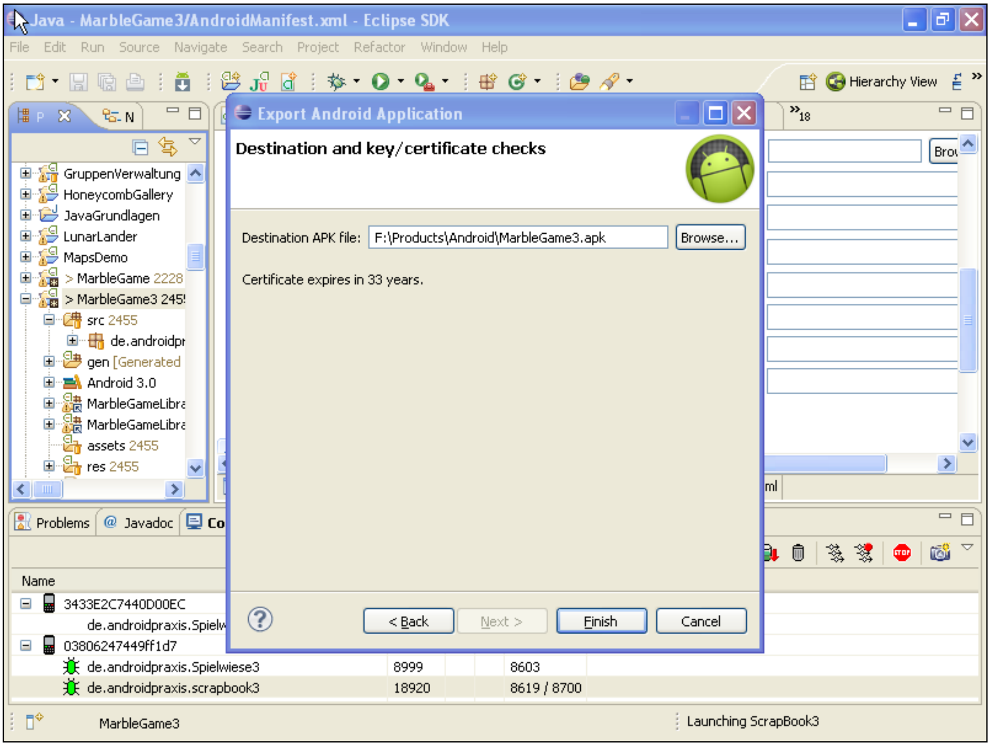


Abbildung 4.21: **Zielverzeichnis und Dateiname des APK**

Jetzt geben wir noch an, wo das APK landen soll, betätigen *Finish*, und fertig ist unsere veröffentlichungsreife Version.

Nun bleibt uns nichts weiter zu tun, als die Anwendung auf einen Markt unserer Wahl hochzuladen, dabei eine ordentliche Beschreibung und schönes Bildmaterial zur Verfügung zu stellen und darauf zu warten, dass wir außerordentlichen Erfolg mit der Anwendung haben.

Wenn wir nun eine weitere Anwendung exportieren wollen oder aber eine neue Version, dann nutzen wir den eben erstellten Keystore und wählen das Zertifikat nach seinem Aliasnamen aus:

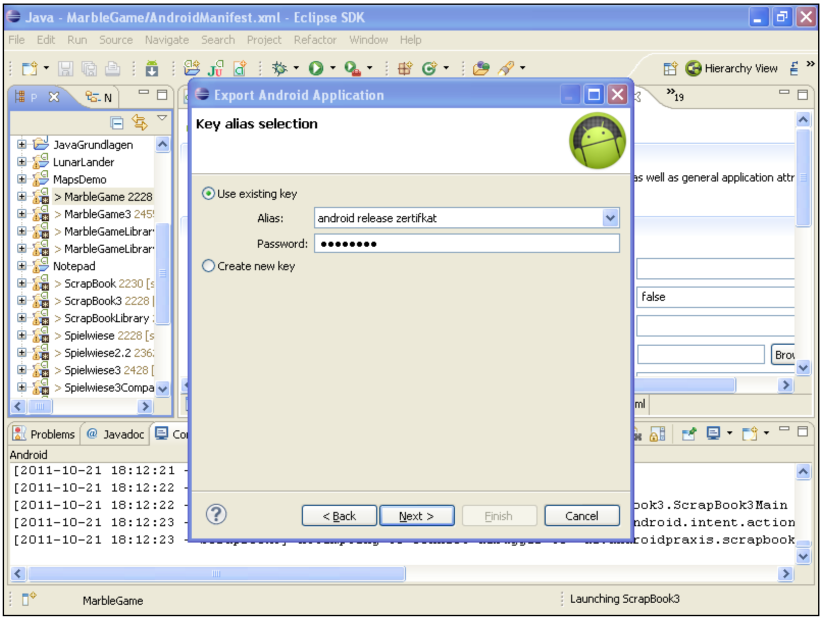


Abbildung 4.22: **Benutzen des zuvor erstellten Zertifikats**

Und so sieht das Hochladen in meinen Google Android Market Account aus, wobei ich diesen Vorgang hier nicht abschließe, da ich eins noch nicht gemacht habe: die Anwendung wirklich fertig programmiert ;-)

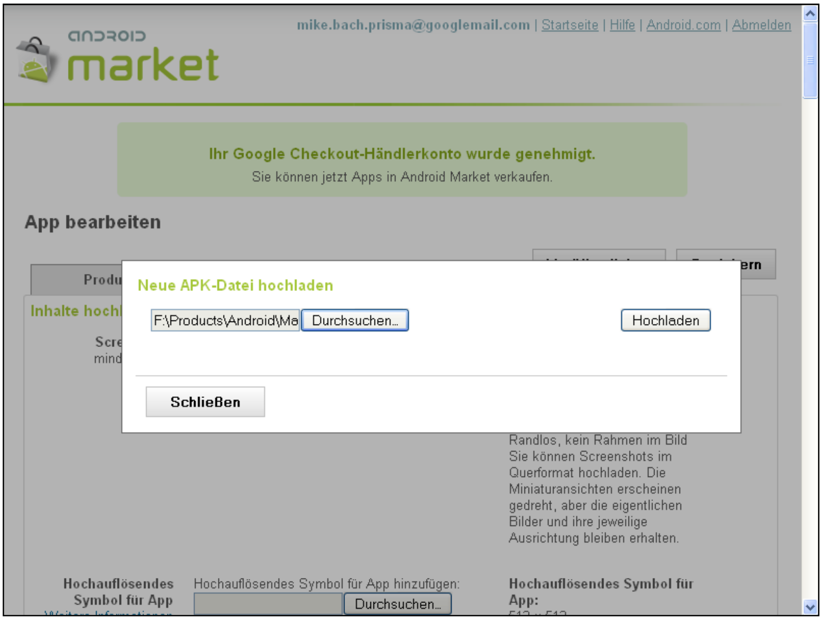


Abbildung 4.23: **Upload-Dialog**

Hier laden wir das APK-File in das Konto hoch.

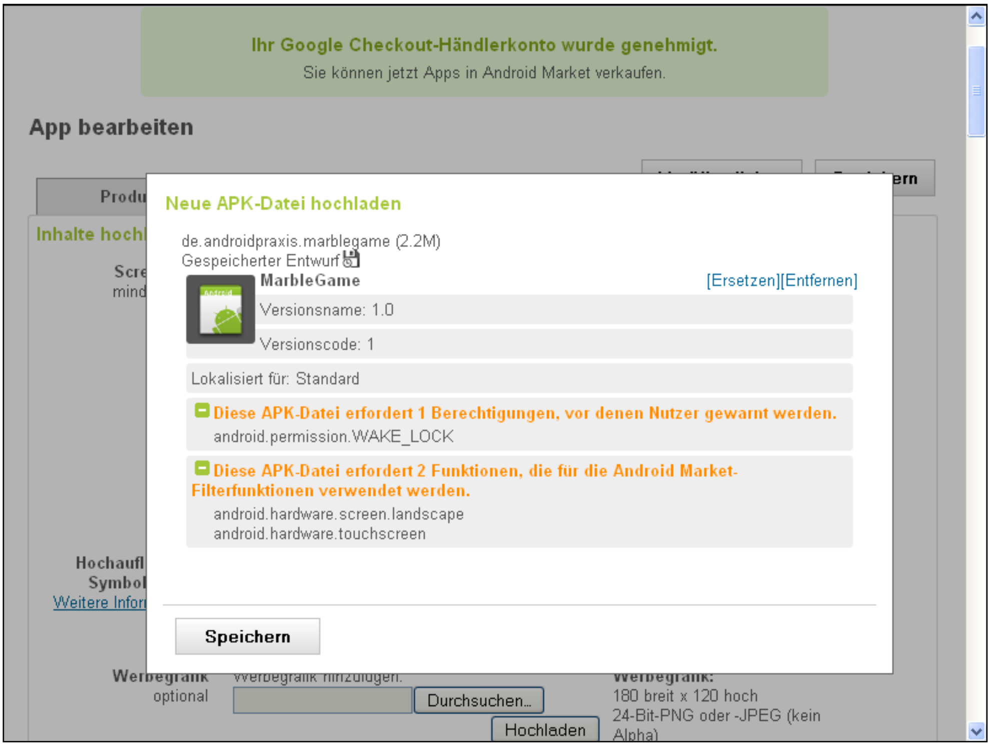


Abbildung 4.24: **Details**

Es werden die Details aus dem Manifest angezeigt (Berechtigungen). Hier z.B. sehe ich, dass ich ein WAKE_LOCK anfordere. Das werde ich wohl rauswerfen, da ich das eigentlich nicht benötige.

Danach müssen wir noch Bilder, Texte und Erklärungen zufügen. Bevor wir allerdings nicht auf »Veröffentlichen« klicken, sieht auch keiner die Anwendung im Markt. Daher kann ich mir damit auch noch Zeit lassen.

Und wenn ich dann demnächst ordentliche Icons und Screenshots fertig habe, werde ich das MarbleGame auch mal in die freie Wildbahn entlassen.

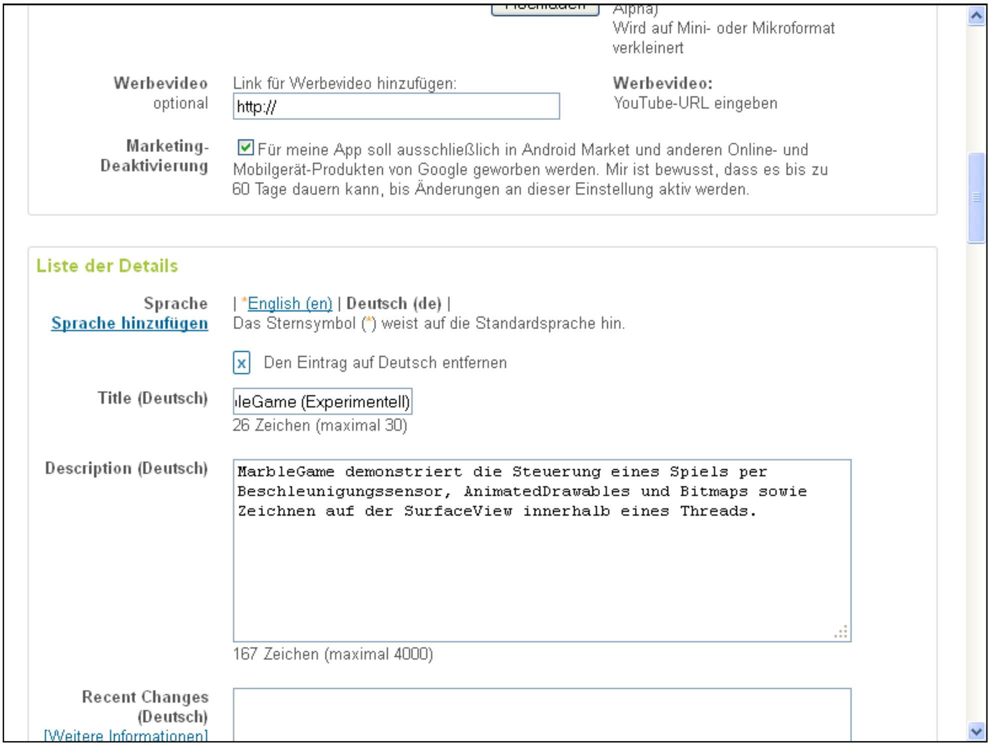


Abbildung 4.25: **Beschreibung und weitere Details erfassen**

4.10 Ein Wort zum Schluss

Wir haben hier nun einige Themen bearbeitet, die Android aus meiner Sicht so spannend machen. Ich hoffe, ich konnte einen Einstieg in die Anwendungsentwicklung bieten, der anspruchsvoll genug ist und Lust auf mehr macht.

Die Kunst, und auch das Leidvolle, beim Schreiben eines Buchs ist, die Balance zwischen der notwendigen Ausführlichkeit und den Grenzen zu finden, die durch Zeit und angestrebten Umfang gegeben sind.

Es mussten einige Themen außen vor bleiben, die für sich genommen auch wichtig und spannend sind, aber den Rahmen komplett gesprengt hätten. Ich hoffe, dass ich Gelegenheit finde, einige dieser Themen in Zukunft zu bearbeiten. Sowohl im Bereich der Grafik als auch im Kommunikationsbereich, der Vernetzung und der Administration von Geräten bieten sich noch vielfältige Möglichkeiten, die es zu nutzen gilt.

Es bleibt mir an dieser Stelle noch zu sagen: danke für eure Aufmerksamkeit, und in Anlehnung an einen Satz, den ich über den kürzlich verstorbenen, unglaublich kreativen – vielleicht auch schwierigen – Steve Jobs gelesen habe: »Bleibt neugierig, bleibt hungrig!«

A Überblick über die Beispielprojekte



Auf der beiliegenden CD befindet sich ein Eclipse-Workspace mit den Beispielprojekten.

Der Workspace kann einfach auf den Rechner kopiert und in der Eclipse ausgewählt werden.

Der Workspace besteht aus den im Buch verwendeten Beispielen und aus einigen Bibliotheken, die auch als Grundlage für eigene Projekte dienen können.

ApplicationLibrary ListMenuActivity ListMenu Utils JSONSerializable DatabaseProvider DatabaseTable StorageHelper NetworkUtilities	Bibliothek, die einige grundlegende Dinge kapselt, die in den Projekten benutzt werden. Besonders interessant dürfte der prototypbasierte Ansatz der DatabaseProvider-Implementierung sein, die eine gute Grundlage für eigene Content-Provider bietet. Außerdem bieten die Klassen NetworkUtilities des StorageHelper einige nützliche, weiterverwendbare Funktionen für HTTP-Kommunikation und das Speichern von Bitmaps sowie das Feststellen des Status des externen Speichers.
ApplicationLibrary23	Ist zurzeit leer, soll Funktionen kapseln, die erst ab Android 2.3 verfügbar sind.
ApplicationLibrary3	Ist zurzeit leer, soll Funktionen kapseln, die erst ab Android 3 verfügbar sind.
MarbleGame	Die Haupt-Activity des MarbleGame-Beispiels. MarbleGame zeigt einige Techniken, die die Darstellung von Bitmaps und Drawables, Threading und das direkte Zeichnen per SurfaceView betreffen. Läuft ab Android 2.2
MarbleGame3	Die Haupt-Activity des MarbleGame-Beispiels. MarbleGame zeigt einige Techniken, die die Darstellung von Bitmaps und Drawables, Threading und das direkte Zeichnen per SurfaceView betreffen. Hier kommt noch eine Spezialisierung hinzu, die das Loader-Konzept von Android 3 demonstriert, indem die Hintergründe des Levels von der Webseite www.androidpraxis.de nachgeladen werden.

Tabelle A.1: **Beschreibung des Workspace**

MarbleGameLibrary Game GameThread PlaygroundView SandwichLayer Vector NetworkUtility NetworkMessageHandler Body Environment SensorHelper	Grundbibliothek für MarbleGame und MarbelGame3. Besonders interessant sind der GameThread, der eine Surface benutzt, um die Spielewelt zu zeichnen, sowie der SandwichLayer, der den vorbeiziehenden mehrschichtigen Hintergrund ermöglicht. Ebenfalls interessant ist die Klasse Body und GameThread, da sie in Zusammenarbeit die Nutzung einer AnimatioDrawable ermöglichen, die den zwinkernden Ball darstellt.
MarbleGameLibrary3 Game3 LevelLoader	Zusätzliche Funktionalität, um das Loader-Konzept zu demonstrieren. In Verbindung mit den Network-Utilities werden die Bitmaps der Hintergründe von www.androidpraxis.de nachgeladen.
ScrapBook	Die Version des ScrapBooks, die ab Version 2.2 lauffähig ist.
ScrapBook3	Die Version des ScrapBooks, die ab Version 3 lauffähig ist. Das ScrapBook demonstriert Content Provider, Touch-Gesten, Multitouch-Gesten, Grafik, ActionBar, Kamera, Galerie, Tonaufzeichnung, Animation und Netzwerkkommunikation.
ScrapBookLibrary	Die gemeinsame Bibliothek des ScrapBooks
Spielwiese	Die Spielwiese sammelt Projekte, die der Veranschaulichung einzelner Aspekte der Android-Grundlagen dienen. Die Verwendung von Intents sowie von Content Providern und anderen Komponenten kann man hier schrittweise ausprobieren. Diese Version läuft unter 2.3, hier ist auch NFC implementiert.
Spielwiese2.2	Spielwiese für Version 2.2
Spielwiese3	Spielwiese für Version 3, besonders interessant wegen der Fragmente und der ActionBar
Spielwiese3Compatibility	Spielwiese mit Konzepten der Version 3, die durch die Kompatibilitätsbibliothek aber auch auf Version 2 laufen.
SpielwieseLibrary	Die Library sammelt Views, Activities und anderes für die Spielwiese.
SpielwieseLibrary3	Spezielle Klassen für Version 3

Tabelle A.1: **Beschreibung des Workspace (Forts.)**

SystemAndHardwareLibrary CameraView CompassView SensorManagement LocationManagement	Diese Bibliothek beinhaltet eine Sammlung von interessanten Klassen, die sich um die Kamera, die Sensoren und das Location Management (GPS) kümmern. Die CameraView ist recht umfangreich und demonstriert die Nutzung der Kamera in einem eigenen Widget.
SystemAndHardwareLibrary233	Hauptsächlich NFC-Hilfsklassen

Tabelle A.1: **Beschreibung des Workspace (Forts.)**

INFO

Ich stelle diese Bibliotheken und Beispielprojekte unter der Apache 2.0-Lizenz zur Verfügung. Die Werke sind nach bestem Wissen erstellt, ich übernehme aber keine Garantie für die Fehlerfreiheit und keine Zusicherung über die Erfüllung bestimmter Leistungen und Funktionen des Codes. Der Einsatz erfolgt auf eigenes Risiko und zum ureigensten Vergnügen. Wenn ihr Fehler findet, sagt mir bitte Bescheid, fehlende Kommentierungen sind eine Kapitulation vor dem zeitlichen Faktor.

Das Copyright für die Bibliotheken und die Beispielprojekte liegt bei:

Dipl.-Inform. (FH) Mike Bach

mike.bach@prisma-net.de

www.androidpraxis.de

www.prisma-net.de

Stichwortverzeichnis

Symbole

A

B

E

F

G

H

I

J

Q

R

S

T